ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Realisierung und Evaluierung einer mobilen Augmented Reality Engine mit OpenGL für Android

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Uwe Ullrich
uwe.ullrich@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Marc Schickler

2015

Fassung 28. Januar 2015

© 2015 Uwe Ullrich

This work is licensed under the Creative Commons. Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Augmented Reality (kurz: AR) ist eine Technologie, welche zunehmend in mobilen Geräten eingesetzt wird. In dieser Abschlussarbeit wird zur Visualisierung von Informationspunkten (Points of Interests) das Projekt AREA verwendet. AREA blendet Informationspunkte in Echtzeit über die Aufnahme der Geräte-Kamera. Die AREA Benutzeroberfläche soll mit dem Einsatz von Open Graphics Library for Embedded Systems (kurz: OpenGL ES) implementiert und evaluiert werden. Um die Benutzeroberfläche von AREA unter Android mit OpenGL ES zu realisieren, wurde eine Grafikbibliothek namens *GraphyFrame* entwickelt. Der Einsatz von GraphyFrame wird nach zwei unterschiedlichen Kriterien evaluiert. Es wird ein Vergleich, zwischen einer bestehenden Benutzeroberfläche und einer Benutzeroberfläche mit GraphyFrame, bezüglich des Stromverbrauchs durchgeführt. Zusätzlich wird die Performance von AREA evaluiert.

Danksagung

Ich möchte mich bei allen Personen bedanken, welche mich auf dem Weg von der Hauptschule bis zu meinem Hochschulabschluss begleitet haben. Als erstes möchte ich mich bei meinen Eltern für die dauerhafte Unterstützung bedanken. Meine Mutter ist kurz vor Beginn dieser Abschlussarbeit unerwartet und viel zu jung verstorben. Diese Abschlussarbeit ist deshalb meiner Mutter gewidmet. Vielen Dank an alle die mich in dieser schweren Zeit begleiten und mir beistehen.

Ohne meine Lerngruppe hätte ich das Studium nicht ohne größere Probleme überstanden. Es hat immer spaß gemacht mit euch Probleme zu lösen und neue Erkenntnisse zu gewinnen. Danke! Weiter möchte ich mich bei der Firma iTM aus Stuttgart für meine zukünftige Arbeitsstelle bedanken.

Zum Schluss möchte ich mich bei dem Institut für Datenbanken und Informationssysteme bedanken.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation und Problemstellung 1

1.2 AREA 3

1.3 Zielsetzung 3

1.4 Aufbau der Arbeit 4

2 Technische Grundlagen **5**

2.1 Vertex in OpenGL 5

2.2 Geometrische Primitive 6

2.3 Datenaustausch mit OpenGL ES 6

2.4 Grafikpipeline 7

2.5 Bilder pro Sekunde 11

2.6 Texturen 11

2.7 Farben in OpenGL 13

2.8 OpenGL ES 1.x vs. OpenGL ES 2.0 14

3 Anforderungsanalyse **15**

3.1 Funktionale Anforderungen 16

3.2 Nicht-funktionale Anforderungen 17

4 Framework **19**

4.1 Architektur des Frameworks 19

4.1.1 Renderer 21

4.1.2 Kamera 23

4.1.3 Objekte in der Szene und ihre Vertices 26
4.2 Ein Kreis als Objekt 28
4.3 Zeichenschleife 30
4.4 Darstellung von Text mittels OpenGL ES 33
4.4.1 Textur Atlas bestehend aus Zeichen 33
4.4.2 Textur zur Laufzeit generieren 35
4.5 Einsatz von GrapyFrame in AREA 35
4.5.1 Darstellung des Radars 36
4.5.2 Darstellung eines POIs 36
4.5.3 Darstellung von Text am Bsp. der Distanz 37

5 Anforderungsabgleich **41**
5.1 Abgleich der funktionalen Anforderungen 42
5.2 Abgleich der nicht-funktionalen Anforderungen 43

6 Diskussion **45**
6.1 Stromverbrauch bei variabler POI Anzahl 45
6.1.1 Messung 46
6.1.2 Auswertung 46
6.2 Bilder pro Sekunde (BpS) 47
6.2.1 Messung 48
6.2.2 Auswertung 49

7 Fazit **51**
7.1 Probleme und Herausforderungen 51
7.2 Ausblick 53

1 Einleitung

1.1 Motivation und Problemstellung

Augmented Reality *(AR)* ist eine Technologie, welche in Echtzeit virtuell generierte Bilder über ein reales Objekt einblendet [ZDB08]. Eine allgemeine Definition von Azuma [Azu97] beschreibt die folgenden drei Eigenschaften für ein AR-System.

1. Ein AR-System kombiniert reale und virtuelle Objekte in einer realen Umgebung.
2. Ein AR-System schließt reale und virtuelle Objekte zusammen.
3. Ein AR-System wird interaktiv, dreidimensional und in Echtzeit ausgeführt.

Viele Anwendungsdomänen profitieren von der AR-Technologie. AR-Anwendungen werden beispielsweise in der Ingenieurswissenschaft, der Unterhaltungsindustrie oder in der Lehre eingesetzt [ZDB08].

Ein High-End-Smartphones besitzt ein hochwertiges Farbdisplay, Hardware- beschleunigte 3D-Grafik und eine integrierte hochauflösende Digitalkamera. Eine integrierte Digitalkamera ermöglicht die Entwicklung von mobilen AR-Anwendungen für High-End-Smartphones [TCG+08] [SSP+14]. Abbildung 1.1 zeigt eine AR-Anwendung für Android, welche in Echtzeit die Sterne am Himmel analysiert, um dem Benutzer Informationen einzublenden. Nicht jede AR-Anwendung kann für ein High-End-Smartphone realisiert werden. Es gibt AR-Anwendungen, welche spezielle Sensoren oder zu viel Rechenleistung benötigen. Beispielsweise in militärischen oder medizinischen AR-Anwendungen [Soo12].

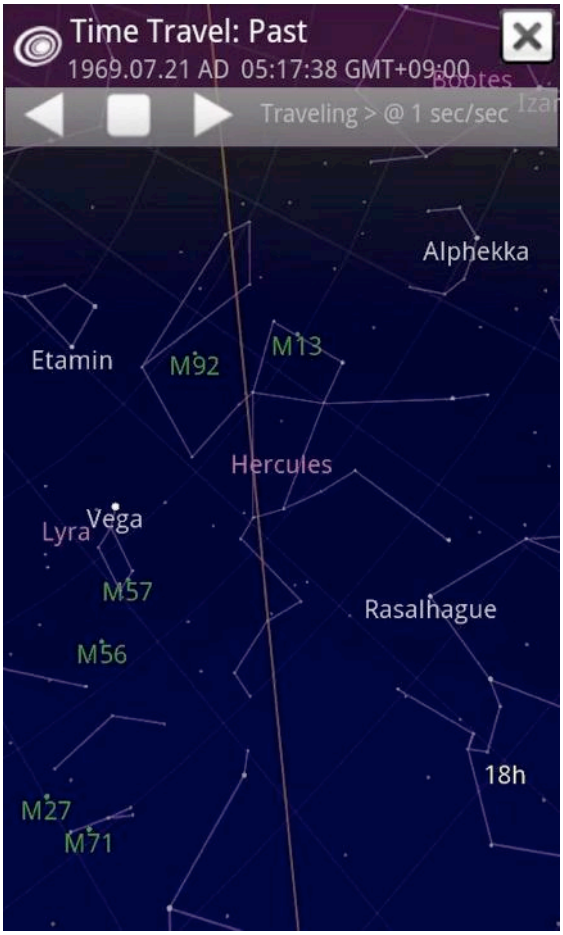

Abbildung 1.1: Google Sky Map ist eine mobile AR-Anwendung. Es wird der Nachthimmel analysiert und Informationen eingeblendet [Dev].

Ein Bestandteil der AR-Technologie ist das Echtzeit-Rendering um grafische Elemente einzublenden. Es gibt AR-Anwendungen mit dem Ziel, dass der Benutzer nicht mehr zwischen Virtualität und Realität unterscheiden kann [BR05]. Das Echtzeit-Rendering von grafischen Elementen kann beispielsweise mit der Open Graphics Library (kurz: OpenGL) realisiert werden. OpenGL ist in der Industrie eine weit verbreitete Programmierschnittstelle zur Darstellung von 3D-Grafiken in Echtzeit [WLH07]. Diese Programmierschnittstelle wird im mobilen Bereich, aber auch bei Simulationen mit Supercomputern eingesetzt. OpenGL ES ist ein 3D-Grafik Standard, welcher auf der OpenGL Bibliothek basiert [SV12]. Die Abkürzung „ES“ steht für Embedded Systems. OpenGL ES wird bei kleinen und portablen Geräten eingesetzt.

1.2 AREA

In dieser Abschlussarbeit wird AREA, eine bereits vorhandene Augmented Reality Engine, eingesetzt. AREA ist die Abkürzung für **A**ugmented **R**eality **E**ngine **A**pplication. In AREA werden alle Points of Interests *(POIs)*, innerhalb eines festgelegten Radius, in einer Live-Ansicht dargestellt. Die Benutzeroberfläche wird in AREA mit grafischen Komponenten aus dem Android SDK gezeichnet [GSP+14]. Beim Zeichnen der Benutzeroberfläche gibt es in AREA gelegentlich Verzögerungen. Die bestehende Benutzeroberfläche in AREA soll ersetzt und mit Hilfe von OpenGL ES dargestellt werden. AREA ist zunächst im Rahmen einer Abschlussarbeit für iOS entwickelt worden. Es wurde jedoch auch eine AREA Version für Android entwickelt. In dieser Abschlussarbeit wird ausschließlich die AREA Version für Android verwendet.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es die grafischen Komponenten in AREA mit OpenGL zu zeichnen. OpenGL soll die Performance der Benutzeroberfläche in AREA optimieren. Im Rahmen dieser Abschlussarbeit wird ein Grafik-Framework entwickelt, welches alle grafischen Komponenten mit OpenGL ES zeichnet. Das in dieser Abschlussarbeit

entwickelte Grafik-Framework wird im Folgenden als *GraphyFrame* bezeichnet. In AREA werden die bestehenden grafischen Komponenten, mit den grafischen Komponenten aus GraphyFrame ersetzt. Der Einsatz von GraphyFrame wird nach der Implementierung in AREA evaluiert. Bei der Evaluierung von AREA wird der Stromverbrauch und die Performance der Benutzeroberfläche ausgewertet.

1.4 Aufbau der Arbeit

In diesem Kapitel wird die AR-Technologie in Kombination mit Echtzeit-Rendering motiviert. Die Ziele für diese Abschlussarbeit werden zusätzlich in diesem Kapitel festgelegt.

In Kapitel 2 werden die technischen Grundlagen und Begriffe von OpenGL vorgestellt. Es werden technische Begriffe für die darauf folgenden Kapitel definiert. Zusätzlich wird jede Stufe der OpenGL Grafikpipeline vorgestellt.

Die Anforderungsanalyse für das Grafik-Framework wird in Kapitel 3 durchgeführt. Dabei werden die funktionalen Anforderungen und die nicht-funktionalen Anforderungen mit Hilfe der bereits vorhandenen AREA Version abgeleitet.

Die wichtigsten Aspekte des Grafik-Frameworks werden in Kapitel 4 vorgestellt. Es wird die Architektur des Grafik-Frameworks an einem Klassendiagramm erklärt. Mit Hilfe eines Sequenzdiagramms wird die Zeichenschleife von dem Grafik-Framework erklärt. Zusätzlich werden verschiedene Vorgehensweisen aufgezeigt, um grafische Primitive zu Zeichnen.

In Kapitel 5 wird festgestellt, ob die Anforderungen erfüllt sind.

Der Einsatz des Grafik-Frameworks wird in Kapitel 6 evaluiert. Es wird der Stromverbrauch protokolliert und ausgewertet. Der Einfluss von den POIs, auf die Anzahl der Bilder pro Sekunde, wird ebenfalls ausgewertet.

In Kapitel 7 werden die Probleme und Herausforderungen noch einmal aufgezeigt. Zusätzlich gibt es einen Ausblick für mögliche Erweiterungen der Benutzeroberfläche in AREA.

2

Technische Grundlagen

In diesem Kapitel werden die Grundlagen von OpenGL vorgestellt. Zum Kern von OpenGL gehört die Grafikpipeline. Die Hauptaufgabe von einer Grafikpipeline ist es ein zweidimensionales Bild zu zeichnen. Das zweidimensionale Bild wird im Allgemeinen aus einer virtuellen Kamera, dreidimensionalen Objekten, Lichtquellen, Texturen, etc. erzeugt. Geometrische Objekte werden mit Vertices beschrieben.

2.1 Vertex in OpenGL

Alle geometrischen Objekte werden in OpenGL mit Hilfe von Vertices beschrieben. Ein Vertex ist ein Punkt im dreidimensionalen Raum. Ein Vertex ist in der Computergrafik eine Ecke von einem Polygon [NFHS12]. In OpenGL ES werden Vertices in einem Array festgelegt. Abbildung 2.1 zeigt beispielhaft die Erstellung eines roten Dreiecks.

```
1 float r = 1, g = 0, b = 0, a = 0;
2 float[] triangle = {
3           0.0f, 0.0f, r, g, b, a //Linker Vertex
4           1.0f, 0.0f, r, g, b, a //Rechter Vertex
5           0.5f, 1.0f, r, g, b, a //Mittlerer Vertex
6 };
```

Abbildung 2.1: Definition eines roten Dreiecks.

2.2 Geometrische Primitive

Geometrische Primitive sind zweidimensionale Einheiten oder Oberflächen. Das können Punkte, Linien oder Dreiecke sein. Ein dreidimensionaler Würfel besteht beispielsweise aus sechs zweidimensionalen Quadraten. Jedes dieser zweidimensionalen Quadrate besteht aus zwei Dreiecken [WLH07]. Ein geometrisches Primitiv wird mit Vertices definiert. Ein Objekt in einer Szene besteht aus beliebig vielen geometrischen Primitiven. In dieser Abschlussarbeit werden die Begriffe geometrisches Primitiv und grafisches Primitiv synonym verwendet.

2.3 Datenaustausch mit OpenGL ES

OpenGL ES wird als native Systembibliothek in Android ausgeführt. Java Code wird in einer Laufzeitumgebung ausgeführt. Unter Android wird diese Laufzeitumgebung als *Dalvik VM* bezeichnet. Die Dalvik VM kann nicht direkt auf eine native Systembibliothek zugreifen. Mit Hilfe des *Java Native Interface* (JNI) kann Java jedoch native Methoden auf einem System aufrufen. Alle Methoden sind in dem Packet des Android SDKs $javax.microedition.khronos.opengles$ mit Hilfe von JNI implementiert. Das Android SDK bietet zusätzlich Java Klassen an, mit denen native Speicherblöcke direkt allokiert und beschrieben werden können. Die native Umgebung und OpenGL ES können auf den nativen Speicher zugreifen. Der native Speicher unterliegt nicht einer automatischen Speicherbereinigung. Bei Android hingegen muss ein Entwickler nicht auf die Speicherbereinigung achten [Kev13].

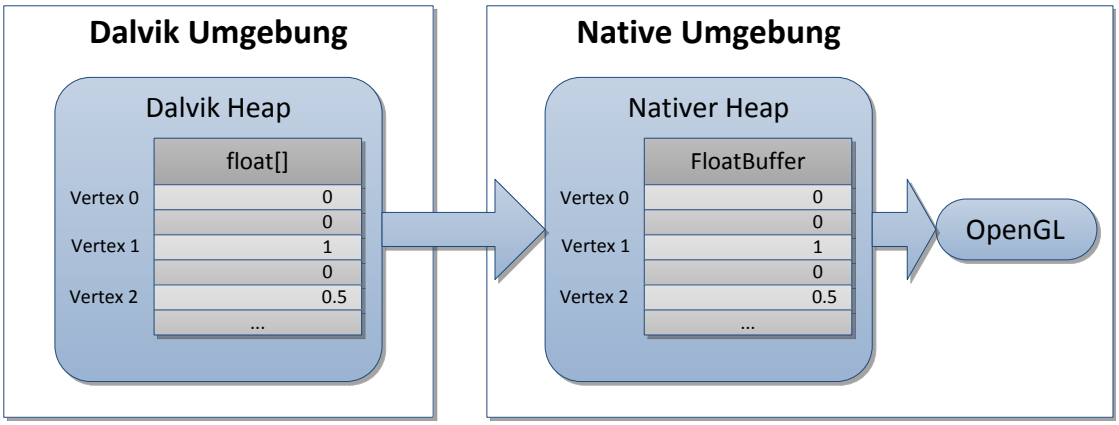

Abbildung 2.2: Datenübertragung der Vertices von der Dalvik VM in die native Umgebung. In Anlehnung an [Kev13].

Abbildung 2.2 zeigt schematisch die Datenübergabe der Vertices in die native Umgebung. OpenGL ES kann somit die benötigten Daten auslesen. Der Float-Datentyp in Java besteht aus 32 Bits. Für jeden Floatwert müssen somit 4 Bytes nativ allokiert werden. Für ein Objekt in der Szene lässt sich die Anzahl der nativen Bytes mit der Formel 2.1 berechnen.

$$\text{Anzahl native Bytes} = \text{Länge Vertice Array} \times \text{Bytes pro Float} \tag{2.1}$$

2.4 Grafikpipeline

Abbildung 2.3 zeigt den Aufbau der OpenGL Grafikpipeline in einer vereinfachten Darstellung. Im Folgenden werden die verschiedenen Stufen anhand ihrer Funktionsweise erklärt. Eine Pipeline besteht im Allgemeinen aus mehreren Stufen [HP96].

- *Applikation:* Das Dreieck in Abbildung 2.3 repräsentiert die Applikations-Stufe. In dieser Stufe wird die dargestellte Szene erstellt oder verändert. Veränderungen in einer Szene können durch Interaktionen des Benutzers ausgelöst werden. In AREA müssen beispielsweise POIs bei einer Drehung des Smartphones rotiert werden. Die Ausgaben der Applikations-Stufe sind die grafischen Primitive, welche später

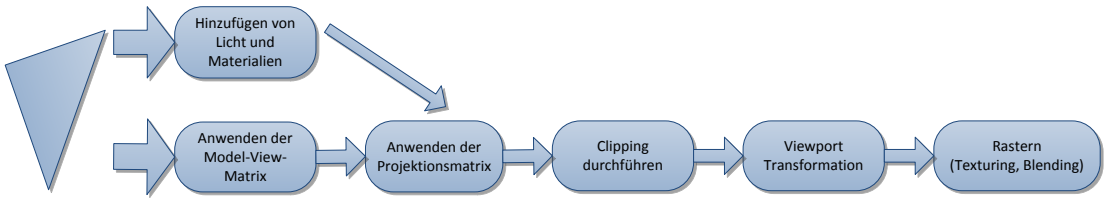

Abbildung 2.3: Abstrakter Aufbau der OpenGL ES 1.1 Rendering Pipeline [ZG12].

auf dem Bildschirm angezeigt werden sollen [AMHH08]. Die Implementierung der Applikations-Stufe kann die Performance der nachfolgenden Stufen beeinflussen. Befinden sich beispielsweise in AREA viele POIs im Sichtfeld, dann müssen entsprechend viele grafische Primitive an die nachfolgenden Stufen übergeben werden.

- *Model-View-Matrix:* Ein Objekt in einer Szene durchläuft verschiedene Koordinatensysteme, bis es schließlich an der richtigen Stelle gezeichnet wird. Jedes Objekt hat am Anfang sein eigenes Objektkoordinatensystem. Mit der Model-Matrix wird das Objekt in dem Weltkoordinatensystem positioniert [AMHH08]. Die Vertices eines Objekts werden dabei mit der Model-Matrix multipliziert [ZG12]. Das Weltkoordinatensystem gibt es nur einmal. Nach dem alle Objekte in einer Szene mit der Model-View-Matrix multipliziert wurden, sind alle Objekte in dem selben Weltkoordinatensystem positioniert. In einer Szene wird, zusätzlich zu den Objekten, eine virtuelle Kamera positioniert. Die View-Matrix transformiert die Kamera in den Ursprung, mit dem Blick in Richtung positiver Z-Achse. Das erleichtert das Clipping in einer späteren Stufe der Grafikpipeline. Die Model-Matrix und die View-Matrix werden mit Hilfe der Matrixmultiplikation zu einer Model-View-Matrix zusammengefasst [AMHH08].

- *Projektionsmatrix:* In der Projektion-Stufe wird das Sichtvolumen mit der Projektionsmatrix in einen Einheitswürfel transformiert. Das Sichtvolumen ist ein unsichtbarer Körper in der Szene. Der Inhalt des Sichtvolumens wird auf den Bildschirm projeziert. Abbildung 2.4 zeigt die zwei Arten von Projektionen, welche am häufigsten eingesetzt werden. Bei der orthogonalen Projektion ist das Sichtvolumen

eine rechteckige Box. Das Sichtvolumen bei einer perspektivischen Projektion ist hingegen in der Form eines Pyramidenstupfs [AMHH08]. Abbildung 2.5 zeigt, dass jedes geometrische Primitiv, bei einer orthogonalen Projektion, mit parallelen Linien auf den zweidimensionalen Bildschirm abgebildet wird. Jedes Objekt wird bei einer orthogonalen Projektion, unabhängig von der Entfernung in der Szene, in der gleichen Größe gezeichnet [WLH07]. Bei einer perspektivischen Projektion ist die größe eines Objekts abhängig davon, wie weit es von der Kamera in der Szene entfernt ist. Ein Objekt das weiter von der Kamera entfernt ist, wird kleiner dargestellt. Parallele Linien konvergieren bei der perspektivischen projektion im Horizont. Die orthogonale Projektion und die perspektivische Projektion kann jeweils mit einer $4 \times 4 -$ Matrix realisiert werden. Wenn das Bild am Ende der Grafikpipeline angezeigt wird, ist die Z-Koordinate nicht mehr in dem generierten Bild vorhanden, deshalb wird diese Matrix auch Projektionsmatrix genannt [AMHH08].

- *Licht und Materialien:* Eine Szene wirkt erst dann realistisch, wenn das Aussehen der Objekte ebenfalls beschrieben wird. Um ein Objekt zu beschreiben, werden Materialeigenschaften und Lichtquellen definiert. Der Vorgang bei dem das Wirken von Licht auf das Material bestimmt wird, bezeichnet man als *Schattierung*. Bei der Schattierung werden an verschiedenen Punkten Schattierungsgleichungen berechnet. In den Vertices können Farbinformationen und beliebige weitere numerische Informationen für die Berechnung der Schattierungsgleichungen gespeichert werden. Die Ergebnisse dieser Stufe werden zum Interpolieren in die Raster-Stufe weitergeleitet [AMHH08].

- *Clipping:* Es werden nur grafische Primitive an die Raster-Stufe weitergeleitet, wenn sie entweder vollständig oder teilweise innerhalb des Sichtvolumens liegen. Grafische Primitive außerhalb des Sichtvolumens werden nicht an die Raster-Stufe weitergeleitet, da diese nicht auf dem Bildschirm gezeichnet werden. Bei dem Clipping werden die grafischen Primitive zugeschnitten, welche nur teilweise innerhalb des Sichtvolumens liegen. Hat eine Linie beispielsweise ein Vertex innerhalb und ein Vertex auserhalb des Sichtvolumens, dann ersetzt ein neuer Vertex, welcher am Schnittpunkt mit dem Sichtvolumen liegt, den außerhalb liegenden Vertex. Clip-

ping wird bei grafischen Primitiven immer gegen den Einheitswürfel durchgeführt [AMHH08].

- *Viewport-Transformation:* Die Viewport-Transformation sorgt dafür, dass das Bild an einer beliebigen Position auf den Bildschirm gezeichnet werden kann. Das Abbilden auf den Bildschirm ist eine Translation gefolgt von einer Skalierungsoperation. Die daraus resultierenden Koordinaten sind die *Gerätekoordinaten*. OpenGL verwendet ein kartesisches Koordinatensystem, somit liegt der Punkt $(0, 0)$ in der unteren linken Ecke [AMHH08].
- *Rastern:* Das Ziel des Rasterns ist es, die Farbe für die einzelnen Pixel des geometrischen Primitiv zu berechnen und das fertige Bild zu zeichnen. Beim Rastern werden insbesondere auch Texturen und die Transparenz eines geometrischen Primitivs mit berücksichtigt. Das Rastern wird vollständig vom Grafikprozessor durchgeführt.

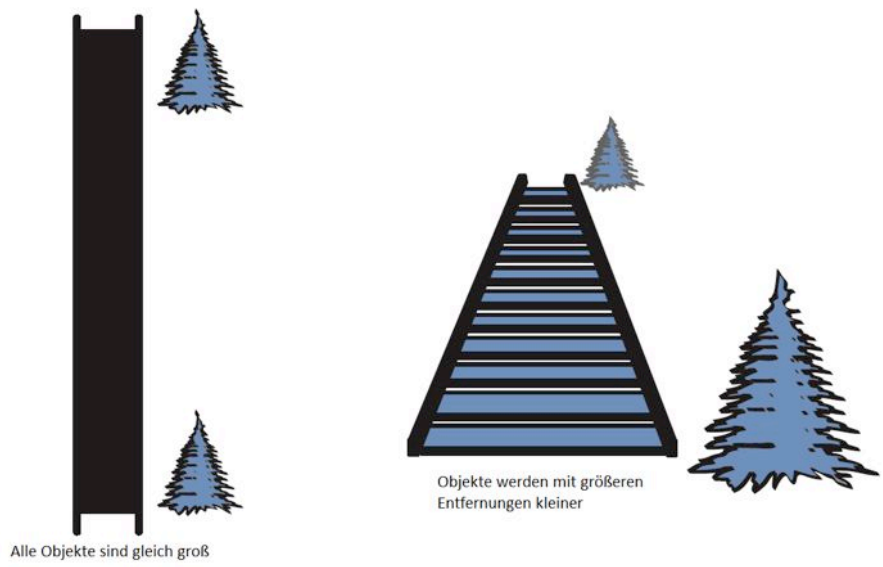

Abbildung 2.4: Orthogonale Projektion vs. perspektivische Projektion [WLH07]

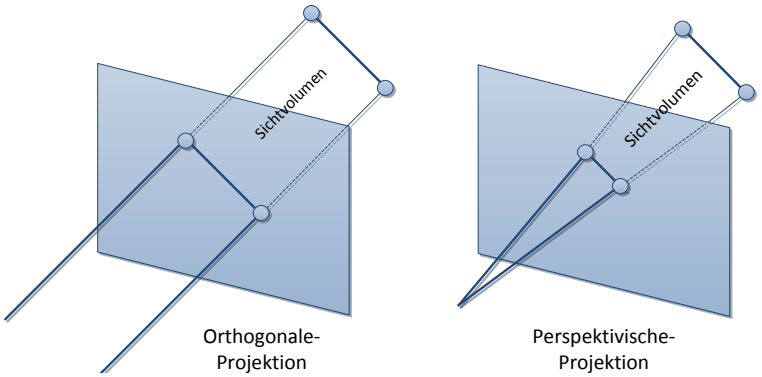

Abbildung 2.5: Sichtvolumen bei orthogonaler Projektion und perspektivischer Projektion. In Anlehnung an [AMHH08].

2.5 Bilder pro Sekunde

Die Geschwindigkeit, in welcher die Bilder gezeichnet werden, wird in Bilder pro Sekunde (BpS) gemessen. Das Maß gibt die Anzahl der gezeichneten Bilder in einer Sekunde an. Die Anzahl der Bilder pro Sekunde hängt von den Berechnungen für jedes Bild ab [AMHH08]. Auf allen bisher verfügbaren Android Geräten ist die maximale Anzahl der BpS auf 60 BpS beschränkt. In der Praxis ist es nicht leicht 60 BpS zu erreichen, da zum Beispiel hochauflösende Tablet-PCs viele Pixel füllen müssen. Es wird dann versucht 30 BpS zu erreichen, um eine flüssige Darstellung einer Szene zu erhalten [ZG12].

2.6 Texturen

Eine Textur ist ein Bild, welches aus *Pixel* besteht. Im Zusammenhang mit Texturen wird ein Pixel auch *Texel* genannt. Texturen besitzen ihr eigenes Koordinatensystem. Im Unterschied zu einem kartesischen Koordinatensystem mit den bekannten (x, y, z) Koordinaten, verwenden Texturen ein Koordinatensystem mit den Koordinaten s und t. Die Texturkoordinaten sind normiert auf das Intervall $[0, 1]$ und haben keine Einheiten [SV12].

Abbildung 2.6 zeigt, dass OpenGL eine Textur immer als ein Quadrat mit der Größe $s = 1$ und $t = 1$ behandelt. Der Ursprung des Quadrats $(0, 0)$ liegt in der unteren linken Ecke und der Punkt $(1, 1)$ liegt in der oberen rechten Ecke [KM13]. Das *UV-Mapping* ist ein Vorgang, bei dem eine Textur auf ein geometrisches Primitiv abgebildet wird.

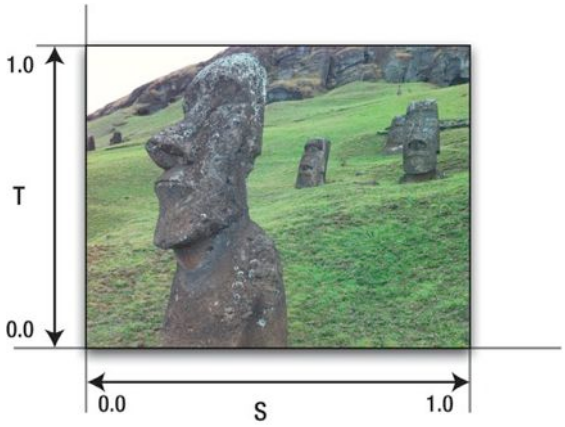

Abbildung 2.6: Texturkoordinaten sind unabhängig von der Textur und immer auf dem Intervall $[0, 1]$ definiert [SV12].

In der Computergrafik wird nur in der OpenGL Domäne von s und t Koordinaten gesprochen. Bei anderen Technologien werden die Texturkoordinaten immer als u und v Koordinaten benannt [SV12]. Um eine Textur zeichnen zu können, muss sie in den Speicher geladen und ihr eine Textur-ID zugewiesen werden. Die Textur-ID wird auch als Texturname bezeichnet. Bei mehreren geladenen Texturen kann mit Hilfe der Textur-ID auf die aktuelle Textur verwiesen werden [KM13]. OpenGL unterstützt viele verschiede Bildformate. In dieser Abschlussarbeit wird ausschließlich das RGBA-Format verwendet. Das RGBA-Format unterstützt mehr als 16 Millionen Farben und Transparenz. Für jeden Farbchannel werden im Speicher jeweil 8 Bit reserviert. OpenGL kann nur Texturen mit den Seitenlängen einer Zweierpotenz verarbeiten [SV12]. Beispielsweise ist ein Bild im PNG-Format mit 32×128 Pixel eine gültige Textur.

Eine Textur wird in OpenGL intern als Array dargestellt. In diesem Array sind die Texel einer Textur gespeichert. In OpenGL werden die Texturen auf geometrische Primitve abgebildet und können somit in verschiedenen Größen auf einem Bildschirm gezeichnet

werden. Bei dem Vorgang des Texturfiltern wird festgelegt, wie ein Texel auf ein Pixel des Bildschirms gefiltert wird. Es wird dabei in drei Fälle unterschieden [lea].

1. *Magnifikation:* Jeder Texel wird auf mehr als einen Pixel abgebildet.
2. *Ohne Filter:* Jeder Texel wird auf genau einen Pixel abgebildet.
3. *Minifikation:* Jeder Texel wird auf weniger als einen Pixel abgebildet.

OpenGL bietet verschiedene Filter für die Magnifikation und die Minifikation an. Die Abbildung 2.7 zeigt, dass die Qualität wie eine Textur gezeichnet wird, je nach ausgewähltem Filter stark variieren kann [lea]. Der Filter *GL_NEAREST* wählt immer den am nächsten gelegenen Texel aus, um den Texel auf einen Pixel abzubilden. Bei dem Filter *GL_NEAREST* sind Pixel auf dem Bildschirm zu sehen. Der Filter *GL_LINEAR* bildet aus den vier am nächsten gelegenen Texel den Durchschnitt und bestimmt somit die Farbe des Pixels. Bei dem Filter *GL_LINEAR* wird die Textur glatt gezeichnet [ZG12].

Abbildung 2.7: Ein Texturfilter hat Auswirkungen auf die Qualität.

2.7 Farben in OpenGL

In OpenGL wird jede Farbe aus einer Mischung von den Komponenten Rot, Grün und Blau dargestellt. Der Alphakanal in eine weitere Komponente, welche für eine transparente Darstellung einer Farbe verantwortlich ist. Jede Komponente ist auf dem Intervall $[0, 1]$ als Float definiert [WLH07]. Bei einem Alphawert von 0 ist ein Pixel völlig undurchsichtig. Analog ist ein Pixel mit einem Alphawert von 1.0 völlig durchsichtig [lea]. Intern wird ein Farbwert auf die am besten passende Farbe abgebildet. Die passende Farbe ist abhängig von dem aktuell verfügbaren Grafikchip. Mit Hilfe der Alphakomponente kann

ein semitransparentes Rot erzeugt werden [WLH07]. Das semitransparente Rot erzeugt den Effekt, als würde man durch rotes Glas schauen. Um einen RBG 255 Wert in einen OpenGL Floatwert zu konvertieren, kann die Formel $Float = \frac{RBG}{255}$ verwendet werden [Mil].

2.8 OpenGL ES 1.x vs. OpenGL ES 2.0

OpenGL ES 1.x hat intern eine *Fixed Function Pipeline*. An einer Fixed Function Pipeline können nur einige Konfigurationen vorgenommen werden. Beispielsweise kann in einer Szene nur eingestellt werden, wie viel Nebel oder Lichter es gibt. Der Vorteil an der Fixed Function Pipeline ist, dass sie leicht zu bedienen ist. Eine Fixed Function Pipeline ist nur begrenzt erweiterbar. Mit OpenGL ES 2.0 wird eine programmierbare Pipeline eingeführt, welche mit Hilfe von *Shadern* programmiert werden kann [Kev13]. Ein Shader kann mit der Programmiersprache OpenGL Shading Language (GLSL) programmiert werden. Die Syntax von GLSL ist stark an C und leicht an C++ angelehnt [SSKLK13]. Die Fixed Function Pipeline wird in OpenGL ES 2.0 nicht mehr spezifiziert. OpenGL ES 2.0 ist deshalb nicht rückwärtskompatibel zu OpenGL ES 1.x [ESV]. Anwendungen sind entweder mit der Fixed Function Pipeline oder mit Shadern implementiert. Die Funktionsweise einer Fixed Fuction Pipeline kann mit Hilfe von Shadern implementiert werden [GPSM14]. Im Allgemeinen ist die Performanz bei OpenGL ES 2.0 besser als bei OpenGL ES 1.x. Der Unterschied in der Performance hängt von den Android Geräten ab. Die OpenGL ES Grafikpipeline wird bei den Herstellern unterschiedlich implementiert. Die OpenGL ES 1.x API bietet einfache Funktionen an, welche es in der OpenGL ES 2.0 API nicht gibt. OpenGL ES 1.x ist für Entwickler ohne OpenGL Erfahrung leichter und schneller zu erlernen [ESV] .

3

Anforderungsanalyse

In diesem Kapitel werden die Anforderungen für die Benutzeroberfläche mit OpenGL ES analysiert. Als Ausgangspunkt für die funktionalen Anforderungen dient die bereits vorhandene Benutzeroberfläche von AREA (siehe Abbildung 3.1). Die POIs sollen an der richtigen Position auf dem Display dargestellt werden. Bei einer Drehung des Smartphones sollen alle POIs rotiert werden. Die Position und die Rotation eines POIs wird von AREA in Echtzeit aus den Sensordaten des Smartphones berechnet [SSP+13]. Das Radar soll alle POIs in einem festgelegten Radius anzeigen. Die Ausrichtung des Radars wird von AREA in Echtzeit berechnet. Unterhalb des Radars ist eine Anzeige für den POI Radius dargestellt. Um alle funktionalen Anforderungen zu erfüllen, reicht der Einsatz von OpenGL ES 1.x aus. Der Einsatz von OpenGL ES 2.0, welcher im Gegensatz zu OpenGL ES 1.x viel mehr Aufwand erfordert, ist nicht notwendig. Der Unterschied zwischen OpenGL ES 1.x und OpenGL ES 2.0 wird in Kapitel 2.8 erklärt.

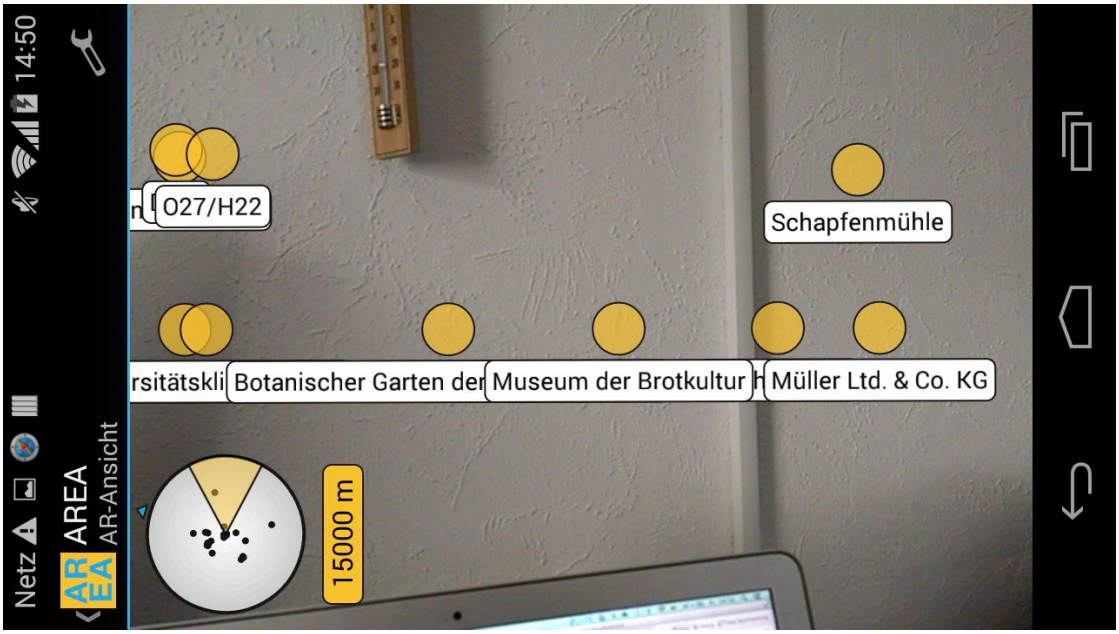

Abbildung 3.1: Grafische Oberfläche von AREA ohne OpenGL ES.

3.1 Funktionale Anforderungen

Im Folgenden werden die funktionalen Anforderungen benannt und beschrieben.

1. *Radar:* Das Radar soll alle POIs im Umfeld anzeigen.
   a) Ein POI soll als schwarzer Punkt auf dem Radar dargestellt werden.
   b) Ein Dreieck soll um das Radar rotieren und in Echtzeit nach Norden zeigen.
   c) Das Sichtfeld soll im Radar als Kreisausschnitt dargestellt werden. Das Sichtfeld rotiert in Echtzeit um den Mittelpunkt des Radars.
2. *POI:* Ein POI soll an der richtigen Position angezeigt werden.
   a) Ein POI soll als Kreis dargestellt werden.
   b) Der Titel eines POIs soll in einer Textbox dargestellt werden.
   c) Ein POI soll in Echtzeit positioniert und rotiert werden.
3. *Distanz:* Die Distanz soll in einer Textbox angezeigt werden.

Die Darstellung von geometrischen Figuren ist mit OpenGL ES keine triviale Aufgabe. Die folgenden funktionalen Anforderungen definieren geometrische Figuren, welche aus geometrischen Primitiven (vergleiche Kapitel 2.2) konstruiert werden.

4. *Kreis:* Ein Kreis soll an einer beliebigen Position dargestellt werden.
5. *Kreisausschnitt:* Ein Kreisausschnitt soll an einer beliebigen Position dargestellt werden.
6. *Textbox:* Eine Textbox soll Text auf dem Display anzeigen.
    a) Im Hintergrund der Textbox soll ein Rechteck mit abgerundeten Ecken dargestellt werden.
    b) Im Vordergrund der Textbox soll ein beliebiger Text dargestellt werden.

3.2 Nicht-funktionale Anforderungen

Im Folgenden werden nicht-funktionale Anforderungen festgelegt.

1. *Stromverbrauch reduzieren:* Der Stromverbrauch soll mit dem Einsatz von OpenGL ES gesenkt werden.
2. *Verzögerungen bei der Darstellung vermeiden:* Während der Benutzung von AREA mit OpenGL ES darf es keine Verzögerungen bei der Darstellung geben. Die POIs sollen ohne sichtbare Unterbrechungen gezeichnet werden.
3. *Anlehnung an bestehende AREA Version:* Die Benutzeroberfläche von AREA mit OpenGL ES soll ähnlich zu der Benutzeroberfläche aus Abbildung 3.1 sein.
4. *Verfügbarkeit:* Die AREA Version mit OpenGL ES soll auf möglichst vielen Android Geräten ausführbar sein.

4

Framework

Um OpenGL ES in AREA einsetzen zu können, ist im Rahmen dieser Abschlussarbeit ein Grafikframework entwickelt worden. In diesem Kapitel werden die wichtigsten Aspekte des Grafikframeworks vorgestellt. Mit dem Einsatz eines Grafikframeworks, muss die Architektur von AREA nicht verändert werden.

4.1 Architektur des Frameworks

Das Grafikframework ist als eigenständiges Eclipse Projekt angelegt und hat den Namen *GraphyFrame*. AREA bindet GraphyFrame ein und verwendet die benötigten Klassen, um die Oberfläche mit OpenGL ES zu zeichnen. Das Ziel von GraphyFrame ist die Abstraktion des Zeichenvorgangs. Abbildung 4.1 zeigt die Architektur von GraphyFrame.

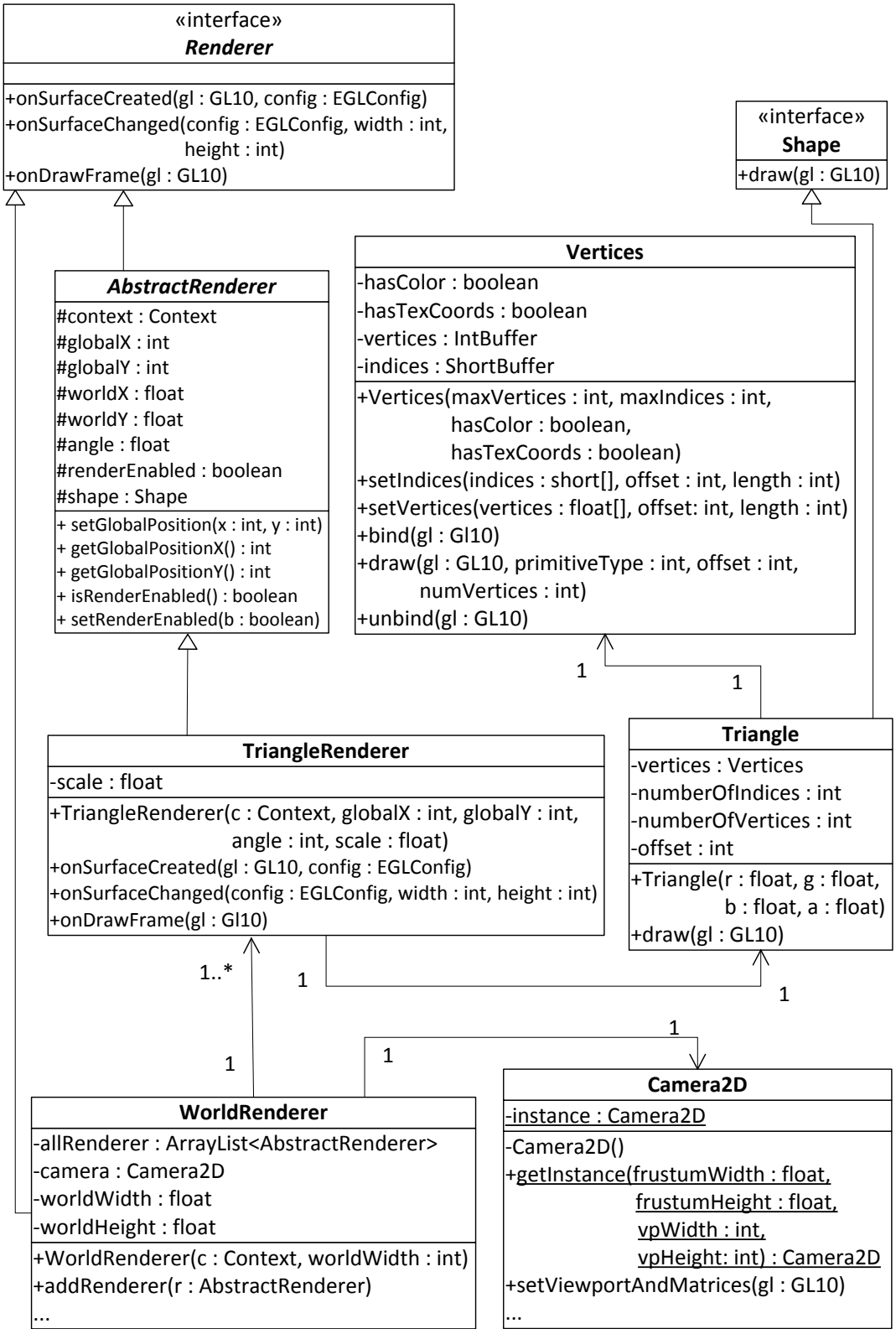

Abbildung 4.1: Die wichtigsten Komponenten der Architektur von GraphyFrame.

Die Klasse *Vertices* ist aus [ZG12] übernommen. Die Klasse *Camera2D* ist an [ZG12] angelehnt.

4.1.1 Renderer

Mit der Klasse *GLSurfaceView* kann eine Zeichenfläche erstellt werden. Auf dieser Zeichenfläche kann man mit Hilfe von OpenGL ES zeichnen. *GLSurfaceView* ist seit Android 1.5 im Android SDK enthalten. Um mit OpenGL ES zu zeichnen, muss eine Listener-Schnittstelle implementiert und bei *GLSurfaceView* registriert werden. Diese Schnittstelle hat die Bezeichnung *Renderer* und besteht aus den folgenden drei Methoden (vergleiche Abbildung 4.1).

- *onSurfaceCreated(...):* Diese Methode wird beim Start der *Activity* aufgerufen. Zusätzlich wird diese Methode aufgerufen, wenn die *Activity* einen Pause-Zustand verlässt.
- *onSurfaceChanged(...):* Diese Methode wird bei jeder Größenänderung der Zeichenfläche aufgerufen. Speziell bei dem Maximieren einer Applikation.
- *onDrawFrame(...):* Diese Methode wird maximal etwa 60 mal pro Sekunde aufgerufen. Alle Zeichenoperationen werden in dieser Methode ausgeführt.

Die Zeichenfläche wird jedes Mal gelöscht, wenn die *Activity* pausiert wird. Wenn die *Activity* wieder fortgesetzt wird, dann wird von *GLSurfaceView* eine neue OpenGL ES Zeichenfläche angelegt und die Methode *onSurfaceCreated(...)* aufgerufen. Alle zuvor festgelegten Zustände in OpenGL ES gehen dabei verloren. Das Problem ist auch als „context loss“ bekannt [ZG12]. Nach jedem „context loss“ werden in GraphyFrame alle notwendigen Texturen neu geladen und an die Grafikpipeline von OpenGL ES übergeben.

Die Architektur von GraphyFrame ermöglicht es individuelle Renderer zu erstellen. Ein Renderer zeichnet ein selbst erstelltes Objekt in der Szene. Ein Objekt kann aus grafischen Primitiven und Texturen bestehen. Der Renderer ist für die Translation, Rotation und Skalierung eines Objekts in der Szene verantwortlich. Das Klassendiagramm in

Abbildung 4.1 zeigt, stellvertretend für weitere Renderer, einen *TriangleRenderer*. Die wichtigsten Rendererklassen in GraphyFrame sind die Folgenden:

- *TextboxRenderer:* Mit OpenGL ES kann nicht direkt ein Text auf dem Bildschirm ausgegeben werden. Es gibt zwei unterschiedliche Verfahren, um dennoch Text zeichnen zu können. Beide Verfahren werden in Kapitel 4.4 genauer vorgestellt.

- *BorderedCircleRenderer:* Mit diesem Renderer ist es möglich einen Kreis mit Rand zu zeichnen. Ein Kreis mit Rand besteht aus zwei überlagernden Kreisen, mit unterschiedlichen Radien.

- *POIRenderer:* Ein POI besteht aus einem Kreis mit Rand und einer Textbox. Der POIRenderer positioniert und dreht die Textbox um den Kreis mit Rand.

- *RadarRenderer:* Der RadarRenderer ist dem WorldRenderer sehr ähnlich. Dem RadarRenderer können dynamisch zur Laufzeit Kreise mit Rändern hinzugefügt werden.

Die Klasse *WorldRenderer* hat Referenzen auf alle Render-Objekte. In dem *WorldRenderer* können zur Laufzeit beliebige Renderer hinzugefügt werden. Der *WorldRenderer* muss sicherstellen, dass es zu keinen Problemen bei mehreren Threads kommt. Es gibt im Wesentlichen zwei unterschiedliche Threads, welche auf den *WorldRenderer* zugreifen. Ein Thread aus AREA erstellt ein Objekt von einem eigens erstellten Renderer. Dieser Thread wird im Folgenden als *AREAThread* bezeichnet. Das erstellte Objekt kann beispielsweise ein POIRenderer sein. Das erzeugte Objekt wird anschließend dem WorldRenderer hinzugefügt. Ein weiterer Thread iteriert in einer Endlosschleife über alle vorhandenen Renderer. Dieser Thread wird im Folgenden als *Zeichenthread* bezeichnet. Die erzeugten Renderobjekte werden in der Arrayliste *allRenderer* (vergleiche Abbildung 4.1) abgespeichert.

Problem: Der AREAThread fügt dem WorldRenderer einen Renderer in die Arrayliste *allRenderer* hinzu. Zum gleichen Zeitpunkt iteriert der Zeichenthread über die Arrayliste *allRenderer*. Das hat zur Folge, dass eine *ConcurrentModificationException* ausgelöst wird.

Lösung: Um eine *ConcurrentModificationException* zu vermeiden, muss sichergestellt werden, dass die Arrayliste während einer Iteration nicht verändert wird. Damit es zu keiner *ConcurrentModificationException* kommt, wird jeder durch den AREAThread neu hinzugefügte Renderer zunächst in einer separaten Arrayliste (nachfolgend als Buffer bezeichnet) gespeichert. Vor jeder Iteration durch den Zeichenthread wird der Buffer geleert. Solange der Buffer geleert wird ist dieser durch ein *Lock* geschützt. Für eine kurze Zeit kann kein neuer Renderer hinzugefügt werden. Das ist jedoch in der Praxis kein Problem. Dieser Lösungsansatz vermeidet ein blockieren des Zeichenthreads. Somit kann dieses Problem keine Verzögerungen bei dem Zeichenvorgang verursachen.

4.1.2 Kamera

In Kapitel 2.4 wird die orthogonale Ansicht und die perspektivische Ansicht erläutert. Um die Anforderungen aus Kapitel 3.1 zu erfüllen, wird in GraphyFrame die orthogonale Ansicht verwendet. GraphyFrame unterscheidet zwischen zwei Koordinatensystemen. In OpenGL wird die Position eines Objekts in Weltkoordinaten festgelegt. AREA positioniert die Objekte in einem Koordinatensystem, bei dem die Anzahl der Pixel in X-Richtung und in Y-Richtung angegeben werden. Im Folgenden wird dies als Globale-Koordinaten bezeichnet. Die Klasse *Camera2D* ist für die Umrechnung zwischen den Koordinatensystemen verantwortlich.

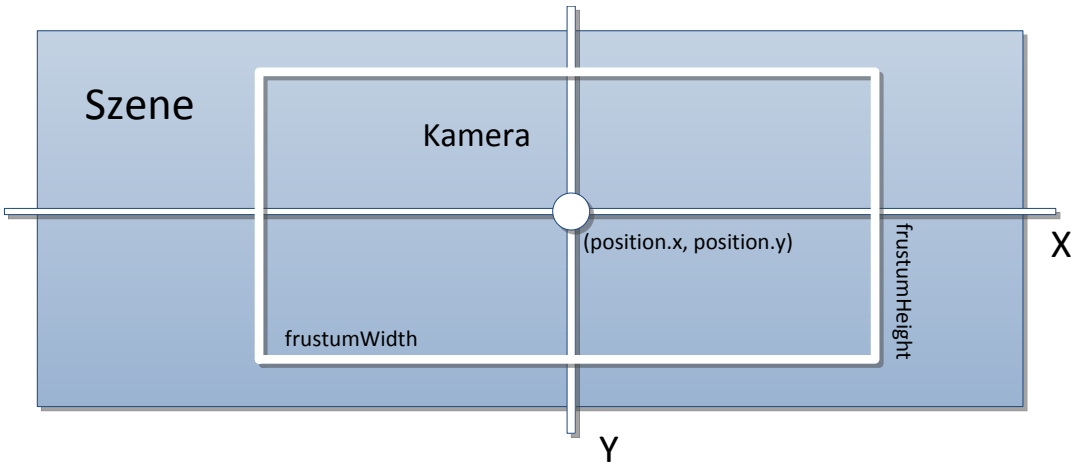

Abbildung 4.2: Das Prinzip der Klasse *Camera2D*.

Alles innerhalb des Sichtfeldes der Kamera wird auf dem Bildschirm angezeigt. In Abbildung 4.2 wird das Prinzip der Klasse *Camera2D* veranschaulicht. Das weiße Rechteck in Abbildung 4.2 ist das Sichtfeld. Bei GraphyFrame wird das Sichtfeld in Z-Richtung auf dem Intervall [-1,1] definiert. Ob ein Objekt in der Szene an dem Punkt (0,0,0) oder an dem Punkt (0,0,1) positioniert wird, macht in der Darstellung auf dem Bildschirm keinen Unterschied. Der Grund dafür ist die orthogonale Ansicht. Im Folgenden wird nicht weiter auf die Z-Koordinate eingegangen, da alle Objekte in der Szene innerhalb des Intervalls [-1,1] in Z-Richtung positioniert sind. Die Kamera kann sich in X-Richtung und in Y-Richtung frei bewegen. Die Kamera wird in AREA mit den Koordinaten (0,0) in der unteren linken Ecke positioniert (siehe Abbildung 4.2). Die Größe des Sichtfelds wird in Weltkoordinaten angegeben. Die Breite des Sichtfelds wird in AREA auf $frustumWidth = 2$ gesetzt (siehe Abbildung 4.2). Somit entspricht die untere rechte Ecke dem Weltkoordinatenpunkt (1,0). Die Höhe des Sichtfelds wird mit der Formel 4.1 berechnet. Somit hat das Sichtfeld ein passendes Verhältnis von der Breite zu der Höhe. Analog könnte man der Sichtfeldhöhe einen konstanten Wert zuweisen und die Sichtfeldbreite ausrechnen.

$$\text{frustumHeight} = \frac{\text{Bildschirmhöhe in Pixel} \times \text{frustumWidth}}{\text{Bildschirmbreite in Pixel}} \tag{4.1}$$

Es existiert in jeder Szene immer genau eine Kamera. Die Klasse *Camera2D* ist deshalb nach dem Singelton Pattern implementiert. Somit existiert immer nur eine Instanz von der Kamera. Die Projektionsmatrix wird mit dem Aufruf der Methode aus Abbildung 4.3 für eine orthografische Projektion (parallele Projektion) initialisiert.

```
1 GL10.glOrthof(int left, int right, int bottom, int top,
2                 int near, int far)
```

Abbildung 4.3: Initialisierung der Projektionsmatrix.

Mit den Parametern kann das Sichtfeld definiert werden. Das Weltkoordinatensystem in OpenGL ES hat seinen Ursprung in der unteren linken Ecke. Die positive X-Achse zeigt nach rechts, die positive Y-Achse zeigt nach oben und die positive Z-Achse zeigt aus der

Zeichenfläche heraus. Ein Koordinatensystem mit diesen Eigenschaften wird auch als *rechtshändiges Koordinatensystem* bezeichnet. GraphyFrame verwendet das rechtshändige Koordinatensystem von OpenGL ES. Bei vielen anderen Grafikbibliotheken zeigt die positive Y-Richtung nach unten. In OpenGL ES ist das ebenfalls möglich, wenn die Parameter „bottom“ und „top“ in *glOrthof(...)* vertauscht werden [ZG12]. Multipliziert man jeweils einen gleichen Faktor auf die Parameter „left“, „right“, „bottom“ und „top“ bei der Methode *glOrthof(...)* , so hat man die Möglichkeit mit der Kamera zu zoomen. Um eine Kamera nach dem Prinzip von Abbildung 4.2 zu erhalten, muss die Methode *glOrthof(...)* folgendermaßen aufgerufen werden:

```
1 gl.glOrthof(position.x - frustumWidth * zoom / 2,
2             position.x + frustumWidth * zoom / 2,
3             position.y - frustumHeight * zoom / 2,
4             position.y + frustumHeight * zoom / 2,
5             1, -1);
```

Abbildung 4.4: Initialisierung der Projektionsmatrix für das Prinzip der Kamera.

Das Attribut „zoom“ hat je nach Wertebereich einen Einfluss auf das Verhalten des Zooms. Die Fallunterscheidung 4.2 beschreibt das Verhalten des Zooms, in Abhängigkeit von dem Wertebereich.

$$\text{zoom} \begin{cases} \in (0,1), & \text{heranzoomen} \\ = 1, & \text{kein Zoom} \\ > 1, & \text{herauszoomen} \end{cases} \tag{4.2}$$

Die Verschiebung der Kamera in X-Richtung und in Y-Richtung wird in einem „position“ Attribut gespeichert. Das Attribut „position“ ist vom Typ *Vector2D*. Die Klasse *Vector2D* wurde im Rahmen dieser Abschlussarbeit implementiert und ist nicht Teil des Android SDK. Die Klasse *Vector2D* enthält eine X-Koordinate und eine Y-Koordinate vom Typ *float*. Zusätzlich bietet die Klasse *Vector2D* die Möglichkeit eine Addition, eine Subtraktion und eine Multiplikation mit zwei Vektoren durchzuführen. Damit sich die Verschiebung auf den Mittelpunkt der Kamera bezieht, wird die Sichtfeldbreite und die Sichtfeldhöhe durch

zwei dividiert. Die linke Begrenzung des Sichtfelds berechnet sich aus der Differenz von der X-Position und der halben Sichtfeldbreite. Die untere Begrenzung des Sichtfelds berechnet sich aus der Differenz der Y-Position und der halben Sichtfeldhöhe. Analog wird die obere und die rechte Begrenzungen für das Sichtfeld berechnet.

4.1.3 Objekte in der Szene und ihre Vertices

Ein Objekt in der Szene besteht aus grafischen Primitiven. Jedes Objekte in der Szene wird in einem eigenen Modellkoordinatenystem modelliert. Die Vertices für ein Objekt in der Szene sind in einer separaten Klasse definiert. Diese Klasse muss die Schnittstelle *Shapes* implementieren. In GraphyFrame sind alle Klassen, welche ein Objekt modellieren im Packet *com.ullrich.shapes* zusammengefasst. In Abbildung 4.1 wird beispielsweise für verschiedene Objekte ein einfaches Dreieck implementiert. Die Klasse *Triangle* implementiert die Schnittstelle *Shape* und übergibt die definierten Vertices an die Klasse *Vertices*.

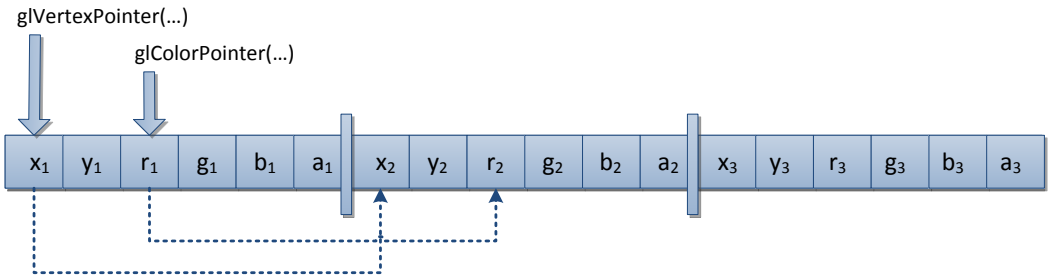

Abbildung 4.5: Unterschiedliche Zeiger im Speicherbereich eines Vertice-Arrays. Der Speicherbereich enthält ein farbiges Dreieck. In Anlehnung an [ZG12].

In einem Array werden Vertices, Texturkoordinaten und Farbwerte sequenziell abgespeichert (siehe Abbildung 4.5 und Abbildung 4.6). Die Klasse *Vertices* übergibt dieses Array mit Vertices und das Array mit den Indizes an die Grafikpipeline. Das Array mit den Indizes bestimmt die Reihenfolge für die Anwendung der Vertices. Die Vertices werden mit der Methode *bind(...)* aus der Klasse *Vertices* an die Grafikpipeline übergeben. Enthält das Vertice-Array Texturkoordinaten oder Farbwerte, dann werden die entsprechenden Pointer in OpenGL ES von der *Vertices* Klasse angepasst.

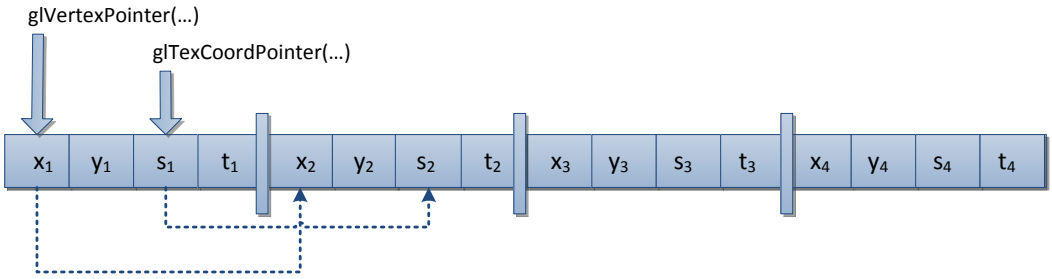

Abbildung 4.6: Unterschiedliche Zeiger im Speicherbereich eines Vertice-Arrays. Der Speicherbereich enthält ein Rechteck mit einer Textur. In Anlehnung an [ZG12].

In Abbildung 4.5 wird der Speicherbereich eines farbigen Dreiecks dargestellt. Nach Formel 2.1 lässt sich die Anzahl der zu allokierenden Bytes für ein farbiges Dreieck berechnen. In der Abbildung 4.5 werden $(2 + 4) \cdot 3 \cdot 4 = 72$ Bytes allokiert. Der Vertex-Zeiger zeigt auf die Positionskoordinaten des Vertice-Arrays. Der Farb-Zeiger zeigt auf die Farbwerte im Speicherbereich. Möchte man ein einfarbiges Dreieck zeichnen, so muss die folgende Bedingung gelten:

$$r_1 = r_2 = r_3 \text{ und } g_1 = g_2 = g_3 \text{ und } b_1 = b_2 = b_3 \tag{4.3}$$

Gilt die Bedingung 4.3 nicht, dann wird ein Dreieck mit einem Farbverlauf gezeichnet. Der Farbverlauf ist von den definierten RGB-Werten abhängig.

Abbildung 4.6 zeigt den Speicherbereich eines Rechtecks mit einer Textur. OpenGL ES bietet in keiner Version die Möglichkeit ein Rechteck mit nur einem Methodenaufruf zu zeichnen. Ein Rechteck kann jedoch in OpenGL ES aus zwei Dreiecken gezeichnet werden (siehe Abbildung 4.7). Im Speicherbereich muss dafür ein Vertice-Array mit vier Vertices gespeichert werden (siehe Abbildung 4.6). Um zwei Dreiecke zu zeichnen, sind normalerweise sechs Vertices notwendig. OpenGL bietet jedoch die Möglichkeit eine Reihenfolge der Vertices festzulegen. Die Reihenfolge wird mit einem Indice-Array festgelegt. Die Anzahl der benötigten Vertices kann somit bei einem Rechteck auf vier Vertices reduziert werden. Im Indice-Array für das Rechteck wird festgelegt, dass bestimmte Vertices doppelt verwendet werden. Abbildung 4.7 zeigt, dass die Vertices v_0 und v_2

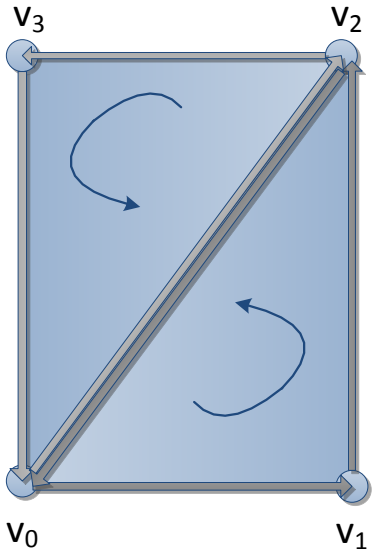

Abbildung 4.7: Ein Rechteck bestehend aus zwei Dreiecken.

zweimal verwendet werden. Diese werden in der Reihenfolge $\underbrace{v_0, v_1, v_2}_{\text{rechtes Dreieck}}$, $\underbrace{v_2, v_3, v_0}_{\text{linkes Dreieck}}$ benutzt, um die zwei Dreiecke aus Abbildung 4.7 zu zeichnen.
Die zwei blauen Pfeile in Abbildung 4.7 zeigen, dass ein Dreieck in OpenGL ES immer gegen den Uhrzeigersinn gezeichnet wird. In Abbildung 4.6 sind zusätzlich noch Texturkoordinaten im Vertice-Array gespeichert. Die Grundlagen von Texturen werden in Kapitel 2.6 erklärt.

4.2 Ein Kreis als Objekt

Im Folgenden wird die Konstruktion eines Kreises in GraphyFrame erklärt. Die Abbildung 4.8 zeigt den theoretischen Aufbau eines Kreises in OpenGL ES. In OpenGL ES wird der Kreis in Modellkoordinaten definiert. Ein Kreis kann mit vielen kleinen Dreiecken angenähert. Werden die Vertices von einem Objekt berechnet, dann wird dies als „Prozedurales Modellieren“ bezeichnet [AMHH08]. Es wird ein Vertex für den Mittelpunkt des Kreises an der Position (0,0) festgelegt. Die Positionen der Randvertices werden mit der Formel $(x, y) = (sin(\alpha) \cdot radius, cos(\alpha) \cdot radius)\ \forall \alpha \in \{0, .., 359\}$ berechnet. Die

Dreiecke, welche den Kreis annähern, werden gegen den Uhrzeigersinn gezeichnet. Das Indice-Array ist folgendermaßen aufgebaut:

$$\underbrace{1,359,0}_{\text{erstes Dreieck}}\ ,\ \underbrace{359,358,0}_{\text{zweites Dreieck}}\ ,\ldots,\ \underbrace{3,2,0}_{\text{359. Dreieck}}\ ,\ \underbrace{2,1,0}_{\text{360. Dreieck}}$$

Aufgrund der vielen Vertices, muss für einen Kreis viel Speicher allokiert werden. Zusätzlich zu der Position eines Vertex, werden bei einem Kreis auch Farbkoordinaten gespeichert. Nach Formel 2.1 werden insgesamt $(2+4)\cdot(359+1)\cdot 4 = 8640$ Bytes für einen Kreis allokiert.

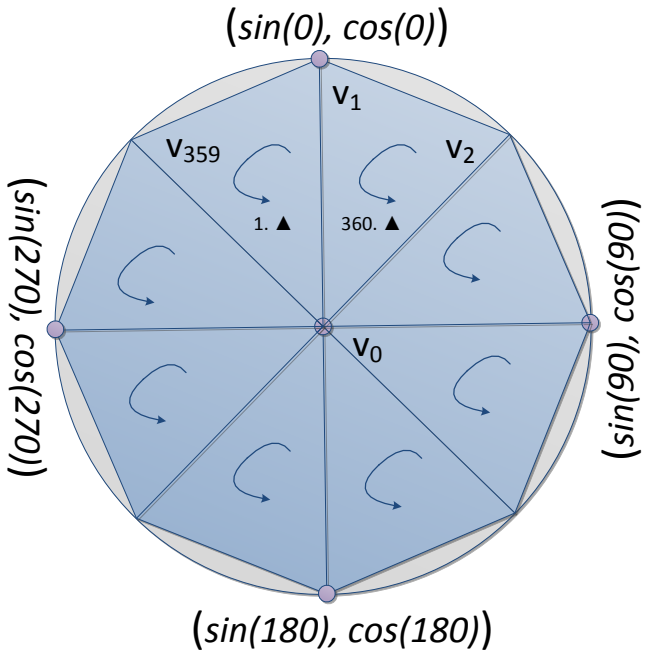

Abbildung 4.8: Theoretischer Aufbau eines Kreises mit Vertices.

Alternativ kann ein Kreis auch mit einem *Dreieckfächer* gezeichnet werden. Ein Dreieckfächer ist ebenfalls ein geometrisches Primitiv in OpenGL ES. Bei einem Dreieckfächer gibt es ein Grundvertex, welches von allen Dreiecken benutzt wird. Verwendet man einen Dreieckfächer, dann müssen keine Indices definiert werden, um einen Kreis zu zeichnen. Vergleicht man das grafische Primitiv „Dreieck“ mit dem grafischen Primitiv „Dreieckfächer“, so gibt es verschiedene Vor- und Nachteile. Die Anzahl der Vertices kann mit dem Einsatz von einem Dreieckfächer reduziert werden. Deshalb müssen

weniger Vertices mit der Model-View-Matrix multipliziert werden. Die Auswirkung ist eine verbesserte Performance während des Zeichenvorgangs. Ein Dreieckfächer ist etwas unflexibler zu benutzen als eine Liste mit definierten Dreiecken. In GraphyFrame wird jedoch auf den Einsatz von Dreieckfächern verzichtet. Mit dem Einsatz von Indizes und einer Liste mit Dreiecken wird eine ähnliche Performanz erreicht [ZG12].

4.3 Zeichenschleife

In diesem Abschnitt werden die Methodenaufrufe innerhalb der Zeichenschleife erklärt. Es wird beispielhaft der Zeichenvorgang eines Dreiecks erklärt. Das Prinzip ist für mehrere Objekte in einer Szene analog. Die Methode *onDrawFrame(...)* wird in der Klasse *WorldRenderer* von einem separaten Thread in einer Endlosschleife aufgerufen. Diese Endlosschleife wird als *Zeichenschleife* bezeichnet. Wenn ein Renderer zu *GLSurfaceView* hinzugefügt wird, dann startet der separate Thread die Zeichenschleife automatisch. Die Reihenfolge der Methodenaufrufe innerhalb der Zeichenschleife, ist in Sequenzdiagramm 4.10 dargestellt. Bei jeder Iteration wird zu Beginn in der Klasse *Camera2D* die Projektionsmatrix mit der Einheitsmatrix initialisiert und das Sichtfeld festgelegt. In der Klasse *TriangleRenderer* wird der Model-View-Matrix Stapel manipuliert. Es gibt für die Manipulation des Stapels in OpenGL ES die Befehle *glPushMatrix()* und *glPopMatrix()* [Gro]. Die oberste Matrix auf dem Stapel ist die aktuelle Matrix. OpenGL ES verwendet die aktuelle Matrix, um die Vertices zu transformieren. Jede Matrix unter der aktuellen Matrix wird erst verwendet, wenn diese zur aktuellen Matrix wird. Die Methode *glPushMatrix()* erstellt eine Kopie von der aktuellen Matrix und legt sie auf den Stapel. Die Methode *glPopMatrix()* entfernt die aktuelle Matrix vom Stapel. Somit wird die darunter liegende Matrix als neue aktuelle Matrix gesetzt. In Abbildung 4.9 wird der Aufbau des Model-View-Matrix Stapels in der Klasse *TriangleRenderer* gezeigt. Im Folgenden wird der Model-View-Matrix Stapel aus Abbildung 4.9 erklärt.

In *Zeile 1* wird der Matrix Stapel für die nachfolgenden Matrix Operationen ausgewählt. Nachfolgend wird in *Zeile 2* und *Zeile 3* eine Kopie der aktuellen Matrix erstellt und diese mit der Einheitsmatrix initialisiert. Die Zeichenschleife iteriert über alle angelegten

```
1  gl.glMatrixMode(GL10.GL_MODELVIEW);
2  gl.glPushMatrix();
3  gl.glLoadIdentity();
4
5  gl.glTranslatef(worldX, worldY, 0f);
6  gl.glRotatef(angle, 0, 0, 1);
7  gl.glTranslatef(-worldX, -worldY, 0f);
8
9  gl.glTranslatef(worldX, worldY, 0f);
10 gl.glScalef(scale, scale, 1);
11
12 shape.draw(gl);
13 gl.glPopMatrix();
```

Abbildung 4.9: Beschreibung einer Matrix mit Matrixoperationen und die Integration in den Model-View-Matrix Stapel.

Renderer. Deshalb muss eine Kopie der aktuellen Matrix angelegt werden. Die *Zeilen 5 bis 7* verschieben das Dreieck in den Ursprung des Koordinatensystems. Im Ursprung wird das Dreieck um den Wert *angle* gedreht. Nach der Drehung wird das Dreieck wieder in die Ausgangsposition zurück verschoben. Somit kann das Dreieck an einer beliebigen Position gedreht werden. In *Zeile 9* wird das Dreieck beliebig positioniert. Die Position muss in Weltkoordinaten angegeben sein. Die Größe des Dreiecks wird mit *glScalef(...)* in *Zeile 10* festgelegt. In *Zeile 12* wird die Methode *draw(...)* der dazugehörigen Form aufgerufen. Alle Matrixoperationen, welche bis zum Aufruf der Methode *draw(...)* festgelegt sind, werden auf die definierten Vertices der Form angewendet. In der letzten Zeile wird die aktuelle Matrix, welche in den *Zeilen 3 bis 10* beschrieben wird, von dem Model-View-Matrix Stapel entfernt. Ist die in den *Zeilen 3 bis 10* beschriebene Matrix entfernt, so wird beispielsweise die Matrix aus einem anderen Renderer zu der aktuellen Matrix.

In diesem Beispiel entspricht die Form einem roten Dreieck. Die Klasse *Triangle* definiert die Vertices des roten Dreiecks gemäß Kapitel 2.1. Die Vertices des roten Dreiecks werden einmalig an das *Vertice* Objekt übergeben. Bei jedem Aufruf der Methode *draw(...)* in dem *Triangle* Objekt, werden sequenziell drei Methoden in dem *Vertice* Objekt aufgerufen. Alle drei Methoden in der *Vertice* Klasse steuern direkt die Grafikpipeline an

und werden im Folgenden erklärt. Die Methode *bind(...)* wird in Kapitel 4.1.3 erklärt. Die Methode *draw(...)* legt das zu zeichnenden grafische Primitiv fest. Zusätzlich wird innerhalb der Methode *draw(...)* unterschieden, ob ein Indice-Array gesetzt ist und ruft entsprechend die richtigen Methoden in der Grafikpipeline auf. Die Methode *unbind(...)* macht alle Zustände, welche zuvor mit der Methode *bind(...)* gesetzt wurden, an der Grafikpipeline rückgängig. Eine andere Instanz der Klasse *Vertice* darf keinen alten Zustand der Grafikpipeline übernehmen. Beispielsweise darf kein Zustand für Farbkoordinaten gesetzt sein, wenn ein anderer Renderer danach ein Rechteck mit einer Textur zeichnet.

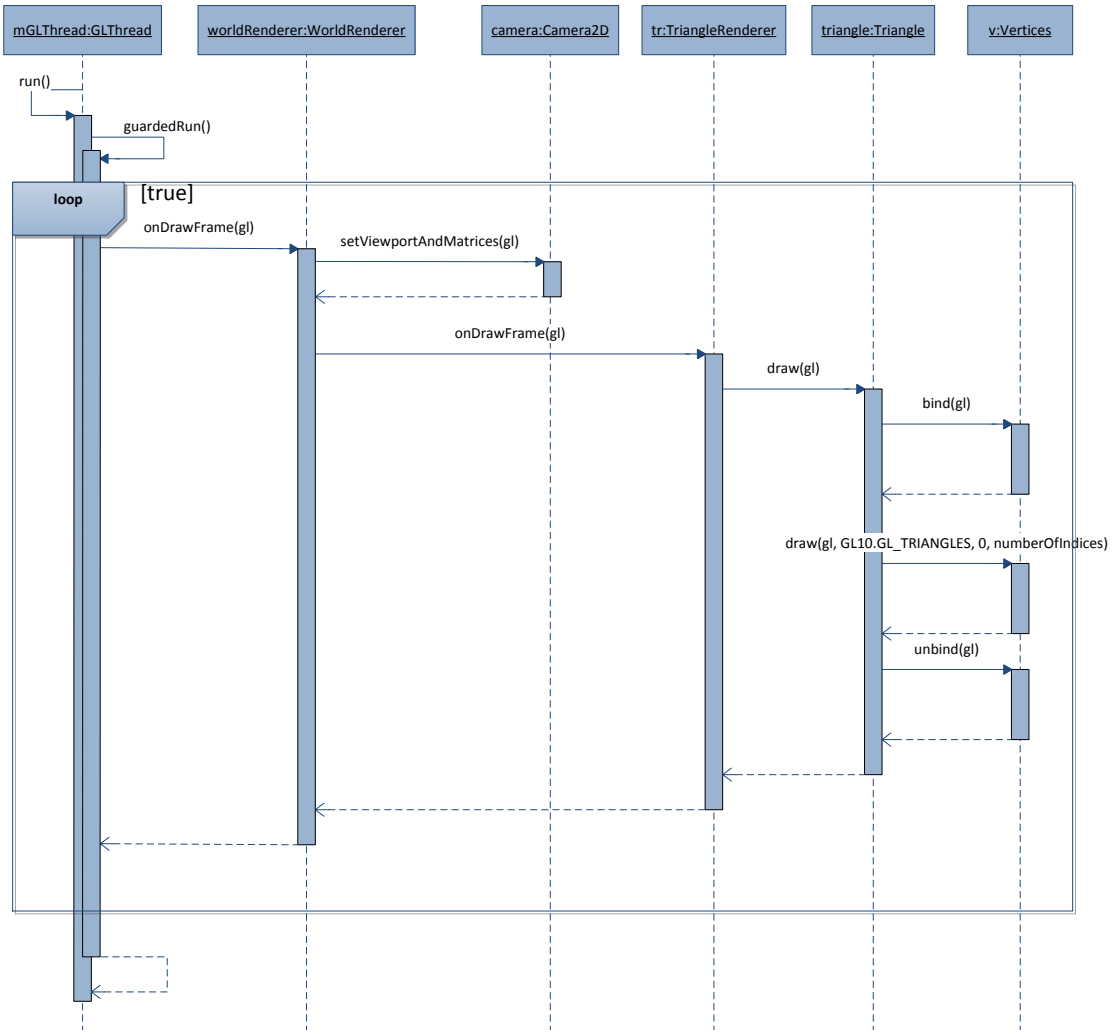

Abbildung 4.10: Aufrufhierarchie der Methoden pro gezeichnetem Frame

4.4 Darstellung von Text mittels OpenGL ES

In OpenGL ES existiert keine Möglichkeit mit nur einem Methodenaufruf einen beliebigen Text zu zeichnen. Um einen beliebigen Text zu zeichnen, muss dieser in Form von einer Textur gezeichnet werden. In diesem Kapitel werden hierfür zwei unterschiedliche Ansätze vorgestellt.

4.4.1 Textur Atlas bestehend aus Zeichen

Das Prinzip in diesem Ansatz ist es einen Text aus mehreren zusammengesetzten Texturen zu zeichnen. Speichert man jedes Zeichen in einer eigenen Textur, dann ist das eine ineffiziente Vorgehensweise Text zu zeichnen, da bei jedem Text mehrere Texturen an die Grafikpipeline übergeben werden. Bei einem langen Text sind deshalb viele OpenGL Aufrufe notwendig. Um einen Text effizient zu zeichnen, muss eine große Textur an die Grafikpipeline übergeben werden. Eine große Textur mit allen notwendigen Zeichen, wird auch als *Textur Atlas* bezeichnet [Wik13].

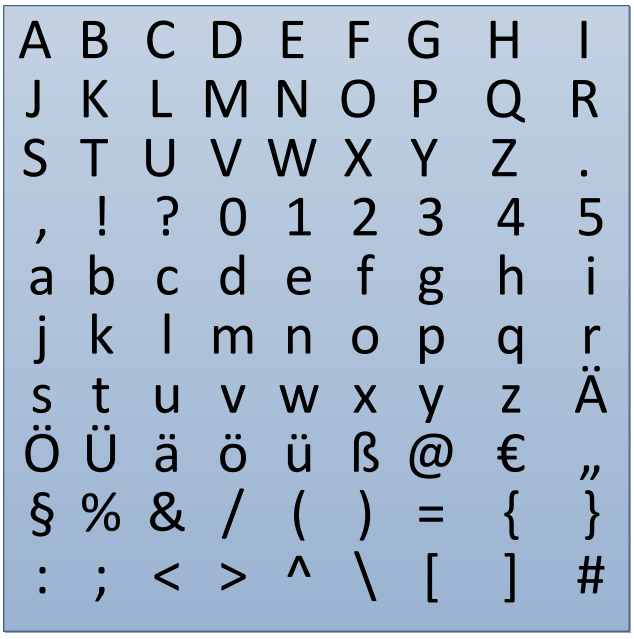

Abbildung 4.11: Aufbau eines beispielhaften Textur Atlas mit verschiedenen Zeichen.

Abbildung 4.11 zeigt den beispielhaften Aufbau von einem Textur Atlas mit Zeichen. Die Breite und die Höhe eines Textur Atlas muss einer zweier Potenz entsprechen (vergleiche Kapitel 2.6). Für jedes Zeichen im Textur Atlas, können die entsprechenden Texturkoordinaten berechnet werden. Im Folgenden werden die Berechnungen der Texturkoordinaten, für ein beliebiges Zeichen aus Abbildung 4.11, beschrieben.

In Abbildung 4.11 befinden sich in einer Zeile neun Zeichen und in einer Spalte zehn Zeichen. Es muss zusätzlich bekannt sein, an welcher Position sich ein Zeichen befindet. Das Zeichen an der Position (0,0) liegt an der unteren linken Ecke. Das Zeichen „U“ befindet sich beispielsweise an der Position (2,7). Zunächst muss die Breite und die Höhe eines Zeichens in Texturkoordinaten berechnet werden (siehe Formel 4.4 und Formel 4.5). Die Breite und Höhe ist bei jedem Zeichen gleich.

$$ZeichenBreiteTexkoord = \frac{(\frac{AtlasBreitePixel}{AnzahlZeichenZeile})}{AtlasBreitePixel} = \frac{1}{AnzahlZeichenZeile} \tag{4.4}$$

$$ZeichenHoeheTexkoord = \frac{(\frac{AtlasHoehePixel}{AnzahlZeichenSpalte})}{AtlasHoehePixel} = \frac{1}{AnzahlZeichenSpalte} \tag{4.5}$$

Allgemein berechnen sich die Texturkoordinaten für ein Zeichen nach den Formeln 4.6 und 4.7. Es muss jeder Eckpunkt für eine Textur festgelegt werden.

$$\begin{aligned} s_{Anfang} &= ZeichenPosX \times ZeichenBreiteTexkoord \\ s_{Ende} &= s_{Anfang} + ZeichenBreiteTexkoord \end{aligned} \tag{4.6}$$

$$\begin{aligned} t_{Anfang} &= ZeichenPosY \times ZeichenHoeheTexkoord \\ t_{Ende} &= t_{Anfang} + ZeichenHoeheTexkoord \end{aligned} \tag{4.7}$$

Das Prinzip um Text mit einem Textur Atlas zu zeichnen, ist auf Grund von den folgenden Nachteilen nicht in GraphyFrame implementiert.

Ein Textur Atlas erfordert die Implementierung einer umfangreichen Logik. Ein Text kann nur begrenzt vergrößert werden, ohne das Pixel zu sehen sind. Es kann kein Textfeld mit runden Ecken nach den funktionalen Anforderungen gezeichnet werden.

4.4.2 Textur zur Laufzeit generieren

In GraphyFrame ist eine Alternative zu dem Textur Atlas implementiert. Eine Textur wird mit Hilfe des Android SDK erstellt und diese an die Grafikpipeline übergeben. Die Klasse *TextBox* aus GraphyFrame generiert eine Bitmap-Objekt. Dem Bitmap-Objekt wird ein *Canvas* [API] hinzugefügt. Die Klasse *Canvas* enthält die Methoden *drawRoundRect(...)* und *drawText(...)*. Eine Textbox besteht in GraphyFrame aus einem beliebigen Text und einem Rechteck mit abgerundeten Ecken im Hintergrund. Für die Darstellung einer Textbox existiert die Klasse *TextboxRenderer* in GraphyFrame. Ein Vorteil bei dieser Alternative ist, dass die Textgröße ohne Qualitätsverlust beliebig angepasst werden kann. Zusätzlich können die Eigenschaften des Rechtecks im Hintergrund beliebig angepasst werden. Ein Nachteil ist, dass mehrere Texturen an die Grafikpipeline übergeben werden. In der Praxis führt das jedoch zu keinen Verzögerungen bei der Darstellung.

4.5 Einsatz von GrapyFrame in AREA

In der bestehenden AREA Version werden alle grafische Elemente mit einem *Canvas* [API] gezeichnet. GraphyFrame enthält alle grafischen Elemente für die Nachbildung der Benutzeroberfläche mit OpenGL. Das Klassendiagramm in Abbildung 4.14 zeigt die Architektur von AREA mit GraphyFrame. Die Klassen *TextboxRenderer*, *AREA-POIRenderer* und *AREARadarRenderer* werden als Render-Klassen bezeichnet. Die Benutzeroberfläche wird aus allen instanziierten Render-Klassen erstellt. Die Klassen *WorldRenderer* und *AREAOpenGLMainActivity* haben Referenzen von allen angelegten Render-Objekten (siehe Abbildung 4.14). Das *WorldRenderer*-Objekt benötigt für die Zeichenschleife alle Render-Objekt Referenzen. Alle Render-Objekte erhalten regelmäßig Daten von einer Instanz der Klasse *AREAOpenGLMainActivity*. Im Folgenden wird der Einsatz der grafischen Elementen aus GraphyFrame vorgestellt.

4.5.1 Darstellung des Radars

Das Radar wird in der bestehenden AREA Version mit der Klasse *AREARadarView* gezeichnet. Diese leitet von der Klasse *android.view.View* ab. In der Klasse *AREA-RadarView* sind alle Daten der POIs in der Liste *locations* gespeichert (siehe Abbildung 4.13). Ein POI wird auf dem Radar als schwarzer Punkt dargestellt. Die Methode *setRadarPoints()* in der Klasse *AREARadarView* berechnet die Positionen von den schwarzen Punkten (siehe Abbildung 4.13). Das Sichtfeld wird auf dem Radar als Kreisausschnitt dargestellt. Der Kreisausschnitt dreht sich um den Mittelpunkt des Radars. In der bestehenden AREA Version wird das Sichtfeld mit der Klasse *AREARadarViewPortView* gezeichnet.

GraphyFrame enthält die Klasse *RadarRenderer*, um ein Radar darzustellen. Die Klasse *AREARadarRenderer* leitet von der Klasse *RadarRenderer* aus GraphyFrame ab (siehe Abbildung 4.13). Das Prinzip die POI-Positionen für das Radar zu berechnen wurde nicht verändert. Mit der Methode *setViewportAngle(int angle)* wird das Sichtfeld für das Radar in der Klasse *RadarRenderer* festgelegt (siehe Abbildung 4.13). Mit dem Einsatz von GraphyFrame muss in AREA keine eigene Klasse für die Darstellung des Sichtfeldes implementiert werden.

4.5.2 Darstellung eines POIs

In der bestehenden AREA Version leitet die Klasse *AREAPointOfInterestView* von der Klasse *android.view.View* ab (analog zu Abbildung 4.13). Eine Instanz der Klasse *AREAPointOfInterestView* zeichnet einen POI. Alle instanziierten POIs werden dem *locationView*-Objekt hinzugefügt. Das *locationView*-Objekt hat den Typ *RelativeLayout*. Mit der Klasse *RelativeLayout* kann ein Layout erstellt werden, welches größer als das Display von einem Smartphone ist. POIs außerhalb des Sichtfeldes sind Bestandteil des *locationView*-Objekts. Bei einer Drehung des Smartphones werden alle Elemente des *locationView*-Objekts in die Gegenrichtung gedreht (siehe Abbildung 4.12). Es muss nicht jeder POI separat gedreht werden. Das Konzept des *locationView*-Objekts erhöht die Performance [GSP+14].

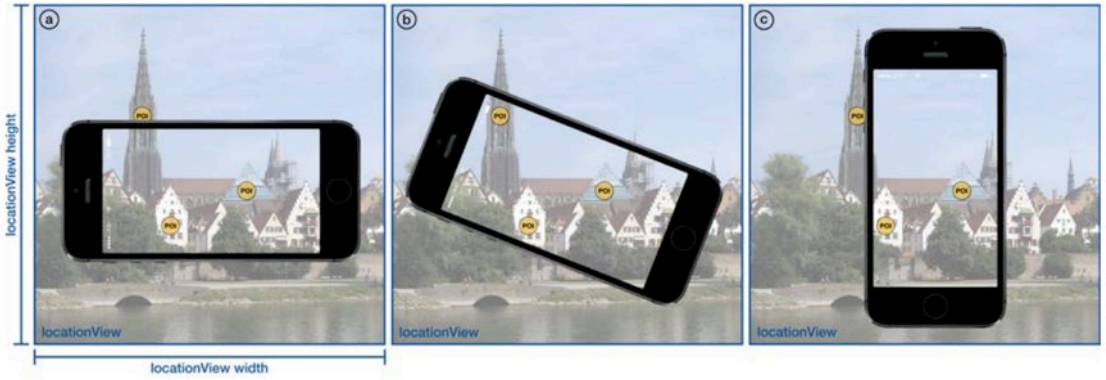

Abbildung 4.12: Prinzip des *locationView*-Objekts [GSP+14].

Mit dem Einsatz von GraphyFrame verändert sich das Konzept POIs in AREA darzustellen. GraphyFrame bietet die Möglichkeit mit der Klasse *POIRenderer* einen POI mit OpenGL darzustellen. In der AREA Version mit GraphyFrame leitet die Klasse *AREAPOIRenderer* von der Klasse *POIRenderer* aus GraphyFrame ab (analog zu Abbildung 4.13). Wenn ein POI das erste Mal im Sichfeld ist, wird eine neue Instanz von der Klasse *AREAPOIRenderer* angelegt und dem *WorldRenderer* mit der Methode *addRenderer(...)* hinzugefügt (siehe Abbildung 4.14). Beim Anlegen eines *AREAPOIRender*-Objekts wird eine Textur generiert. Ein *AREAPOIRender*-Objekt bleibt erhalten, wenn sich der POI nicht im Sichtfeld befindet. Bei einem POI außerhalb des Sichtfeldes wird das Attribut *renderEnabled = false* gesetzt. Ein *AREAPOIRender*-Objekt außerhalb des Sichtfeldes wird nicht gezeichnet. Bei einer Drehung des Smartphones wird jeder einzelne POI separat mit GraphyFrame gedreht. Einzelne grafische Elemente aus GraphyFrame können effizient gedreht und positioniert werden.

4.5.3 Darstellung von Text am Bsp. der Distanz

In der bestehenden AREA Version wird die Distanz mit einer Instanz der Klasse *AREADistanceView* dargestellt. Die Klasse *AREADistanceView* leitet von der Klasse *android.view.View* ab (analog zu Abbildung 4.13). Ein *AREADistanceView*-Objekt wird dem Layout der Benutzeroberfläche hinzugefügt.

GraphyFrame bietet die Möglichkeit mit der Klasse *TextboxRenderer* einen Text darzustellen. In AREA wird keine Unterklasse von der Klasse *TextboxRenderer* benötigt. Es wird ein *TextboxRenderer*-Objekt angelegt und dem *WorldRenderer* mit der Methode addRenderer(...) hinzugefügt.

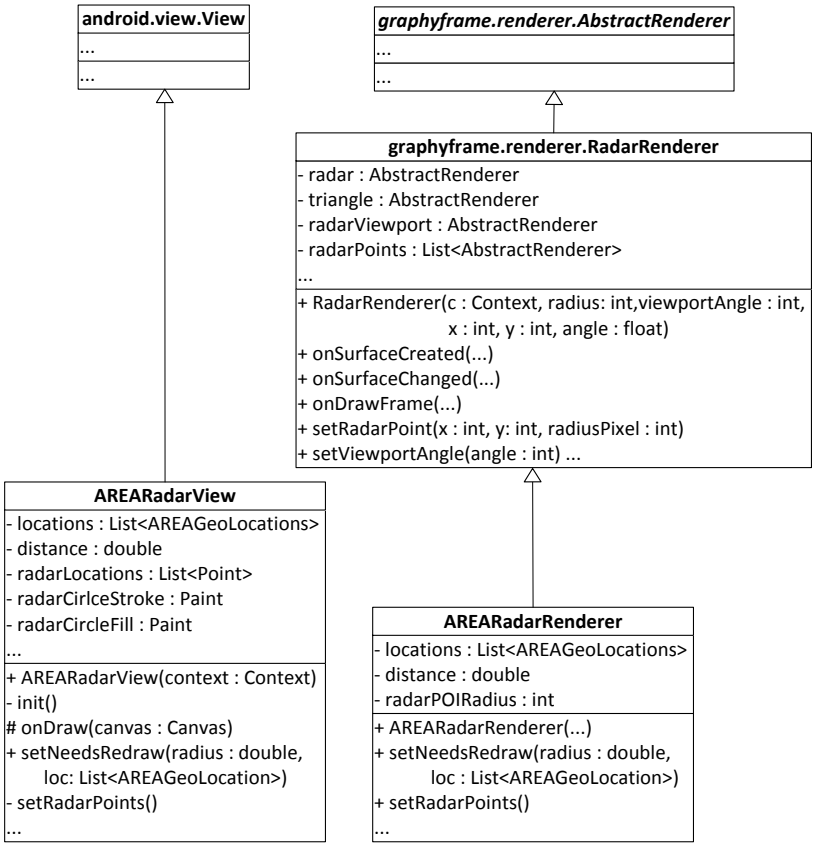

Abbildung 4.13: Vererbungshierarchie des Radars ohne GraphyFrame (links) und mit GraphyFrame (rechts).

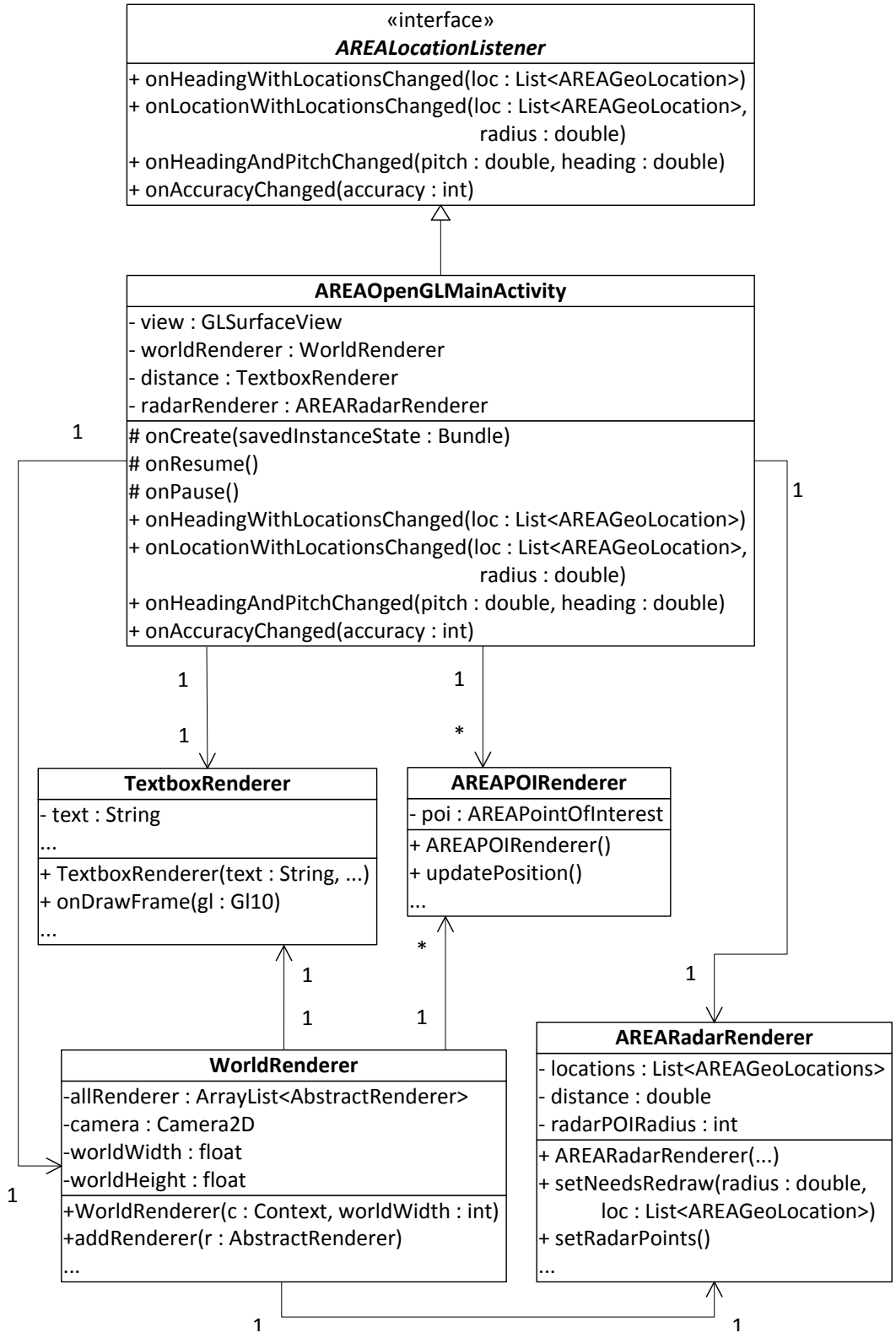

Abbildung 4.14: Architektur von AREA mit GraphyFrame.

5

Anforderungsabgleich

In diesem Kapitel werden die Anforderungen aus Kapitel 3, nach Fertigstellung der Implementierung, abgeglichen. Es wird ermittelt, ob die funktionalen Anforderungen aus Kapitel 3.1 und die nicht-funktionalen Anforderungen aus Kapitel 3.2 erfüllt sind. Abbildung 5.1 zeigt die fertig implementierte Version von AREA mit OpenGL ES. Die MVC-Architektur von AREA ermöglicht es die Benutzeroberfläche von AREA ohne großen Aufwand zu ersetzen. Die AREA Benutzeroberfläche wird mit grafischen Elementen aus GraphyFrame dargestellt. GraphyFrame kann in einem beliebigen Android-Projekt für die Darstellung einer Benutzeroberfläche eingesetzt werden.

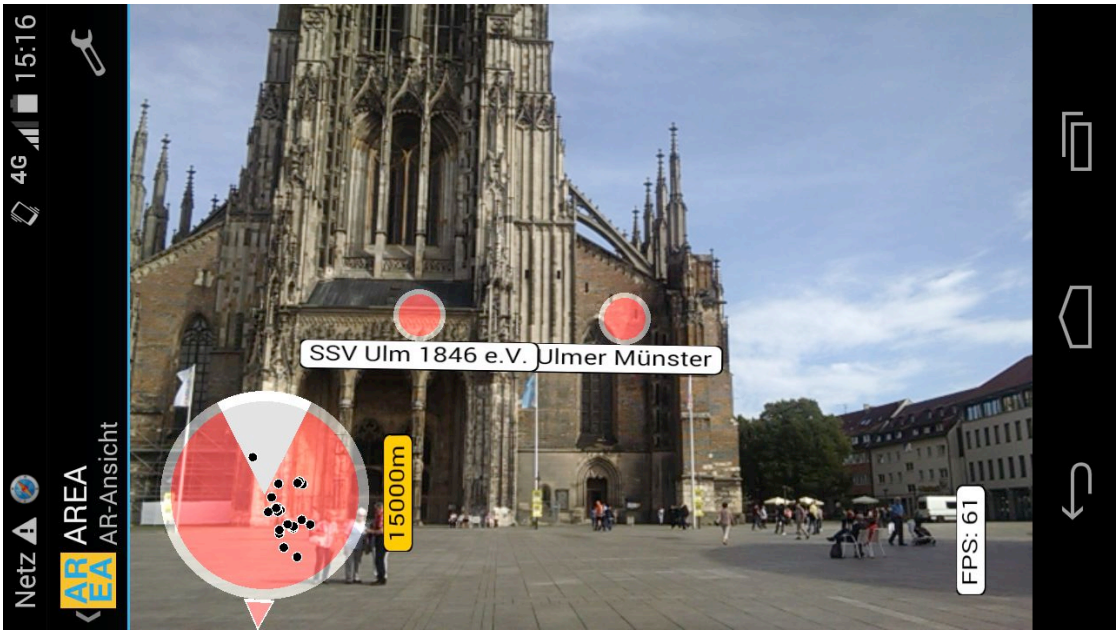

Abbildung 5.1: Grafische Oberfläche von AREA mit OpenGL ES.

5.1 Abgleich der funktionalen Anforderungen

Die Tabellen 5.1 und 5.2 zeigen, dass alle funktionalen Anforderungen aus Kapitel 3.1 erfüllt sind. Zusätzlich wird eine Bewertung für alle funktionalen Anforderungen gegeben.

Nr.	Anforderung	Erfüllt	Bewertung
1	Radar	Ja	Das Radar zeigt alle POIs im Umfeld an. Ein POI wird in Form eines schwarzen Punktes auf dem Radar dargestellt. Das Dreieck in Abbildung 5.1 rotiert um das Radar und zeigt nach Norden. Das Sichtfeld wird in Form eines Kreisausschnittes dargestellt und rotiert um den Mittelpunkt des Radars.

Tabelle 5.1: Abgleich der funktionalen Anforderungen. (1)

Nr.	Anforderung	Erfüllt	Bewertung
2	POI	Ja	Ein POI wird als Kreis dargestellt. Der Titel eines POI wird in einer Textbox unter dem Kreis angezeigt. Alle POIs werden in Echtzeit positioniert und rotiert.
3	Distanz	Ja	Der Radius für die POIs wird unter dem Radar in einer Textbox angezeigt.
4	Kreis	Ja	Ein Kreis kann an einer beliebigen Position dargestellt werden. Ein Kreis ist beispielsweise Bestandteil eines POI.
5	Kreisausschnitt	Ja	Der Kreisausschnitt kann mit den notwendigen Anforderungen gezeichnet werden. In Abbildung 5.1 ist links unten das Radar zu sehen. Der Kreisausschnitt ist ein Teil des Radars.
6	Textbox	Ja	Eine Textbox zeigt einen beliebig Text an. Die Hintergrundfarbe einer Textbox ist beliebig (siehe Abbildung 5.1).

Tabelle 5.2: Abgleich der funktionalen Anforderungen. (2)

5.2 Abgleich der nicht-funktionalen Anforderungen

In der Tabelle 5.3 wird ermittelt, in welchem Umfang die nicht-funktionalen Anforderungen aus Kapitel 3.2 erfüllt sind.

Nr.	Anforderung	Erfüllt	Bewertung
1	Stromverbrauch reduzieren	Ja	Der Stromverbrauch hat sich mit dem Einsatz von OpenGL ES in AREA reduziert. Eine genaue Auswertung gibt es in Kapitel 6.1
2	Verzögerungen bei der Darstellung vermeiden	Teilweise	Wenn AREA gestartet wird, dann treten in den ersten Sekunden Verzögerungen bei der Darstellung auf. Der Grund für die Verzögerungen ist, dass jeder POI zuerst geladen werden muss. Ein POI wird erst dann geladen, wenn er auch in dem Sichtfeld ist. Sind alle POIs geladen, dann kommt es bei normaler Benutzung zu keinen weiteren Aussetzern mehr. Wird die Anzahl der POIs künstlich erhöht, dann sinken auch die BpS. In Kapitel 6.2 ist der Zusammenhang, zwischen Anzahl POIs und den durchschnittlichen BpS in einer Minute, dargestellt.
3	Anlehnung an bestehende AREA Version	Ja	In der AREA Version mit OpenGL ES (siehe Abbildung 5.1) sind alle grafischen Komponenten dargestellt, welche in der AREA Version ohne OpenGL ES (siehe Abbildung 3.1) zu sehen sind. Die AREA Version mit OpenGL ES benutzt jedoch andere Farben und hat zusätzlich noch eine BpS Anzeige integriert. Die Farben können mit Hilfe von Enums geändert werden. Die BpS Anzeige kann in der Klasse *WorldRenderer* deaktiviert werden.
4	Verfügbarkeit	Teilweise	AREA mit OpenGL ES wurde unter der Android Version 4.4.x *(KitKat)* auf drei verschiedenen Smartphones getestet. Alle drei Smartphones hatten unterschiedliche Displaygrößen. Es ist bei allen drei Smartphones zu keinen Fehlern gekommen.

Tabelle 5.3: Abgleich der nicht-funktionalen Anforderungen. (1)

6

Diskussion

In diesem Kapitel wird der Einsatz von OpenGL ES 1.x in AREA evaluiert. Es wird ein Vergleich zwischen den AREA Versionen mit OpenGL ES und ohne OpenGL ES durchgeführt. Bei diesem Vergleich wird der Stromverbrauch beider AREA Versionen ausgewertet. Die grafische Oberfläche der AREA Version ohne OpenGL ES ist mit Hilfe der *View* Klasse aus dem Android SDK implementiert. Die Auswirkungen auf die Anzahl der BpS, bei großer Anzahl von POIs, wird für die AREA Version mit OpenGL ES gemessen.

6.1 Stromverbrauch bei variabler POI Anzahl

In dieser Messung wird der Batteriestatus bei einer unterschiedlichen Anzahl geladener POIs analysiert. Der Verlauf des Batteriestatus wird mit der Klasse *BatteryMeasurement*

protokolliert. Die Messungen sind alle mit einem Motorola Moto X der ersten Generation durchgeführt worden. Während jeder Messung waren in AREA regelmäßig POIs im Sichtfeld.

6.1.1 Messung

In Abbildung 6.1 wird der notwendige Zeitraum dargestellt den Batteriestatus um 5% zu senken. Insgesamt wurden zehn Messungen nacheinander für beide AREA Versionen, bei einer unterschiedlichen Anzahl geladener POIs, durchgeführt. Die Batterie wurde zwischen den Messungen nicht aufgeladen. Am Anfang war die Batterie des Smartphones zu 85% aufgeladen. Es wurde abwechselnd für jede AREA Version eine Messung durchgeführt. Die Anzahl der geladenen POIs wurde nach jeder zweiten Messung erhöht. Nach der Veränderung des Batteriestatus um 5% war eine Messung beendet.

6.1.2 Auswertung

Die AREA Version ohne OpenGL ES benötigt bei zehn geladenen POIs länger den Batteriestatus um 5% zu senken, als die AREA Version mit OpenGL ES. Bei 25, 50, 75 und 100 geladenen POIs benötigt die AREA Version mit OpenGL ES länger den Batteriestatus um 5% zu senken, als die AREA Version ohne OpenGL ES. Die AREA Version mit OpenGL ES hat bei diesen Messungen im Durchschnitt 19 Sekunden länger gebraucht, als die AREA Version ohne OpenGL ES, um den Batteriestatus 5% zu senken. Mit dem Einsatz von OpenGL ES verbraucht AREA somit weniger Strom. Die Messungen werden von den folgenden Einflüssen verfälscht.

- AREA kann nicht bei jeder Messung exakt gleich benutzt werden.
- Die Dauer, wie lange sich die Batterie schon in ihrem Ausgangspunkt befindet, ist bei jeder Messung unterschiedlich.

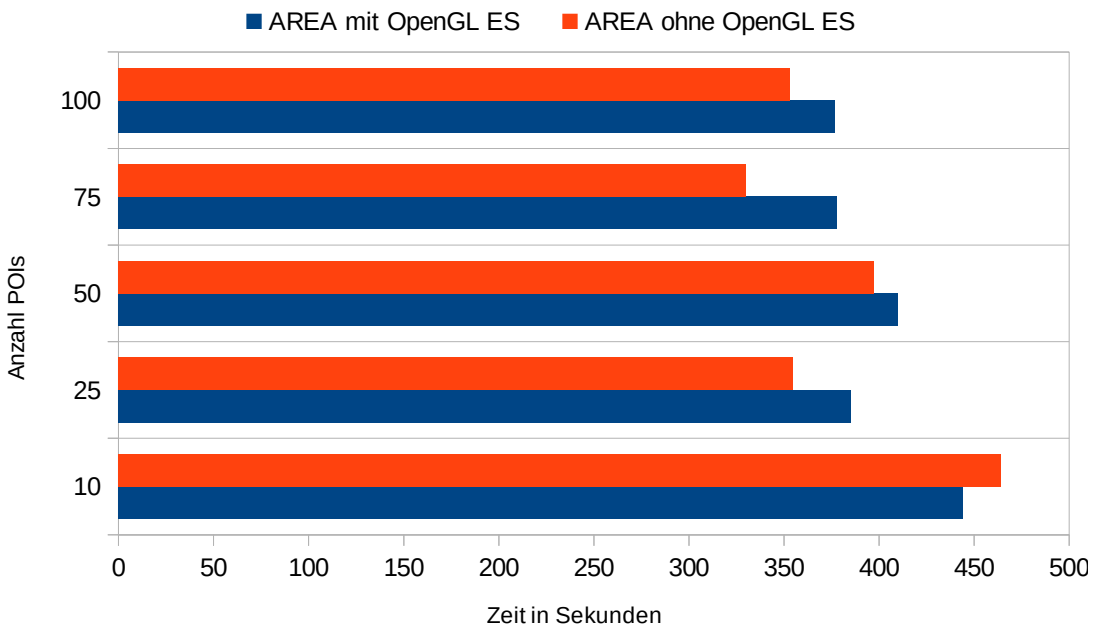

Abbildung 6.1: Messung des Stromverbrauchs bei einer variablen Anzahl von POIs.

6.2 Bilder pro Sekunde (BpS)

Die Anzahl der gezeichneten BpS ist von der Anzahl an geladenen POIs abhängig. Bei jedem Zeichenaufruf wird mit Hilfe der Klasse *FPSCounter* ermittelt, wie viele Zeichenaufrufe es innerhalb einer Sekunde gegeben hat. In der *FPSCounter* Klasse wird bei jedem Zeichenaufruf eine globale Zähl-Variable inkrementiert. Nach einer Sekunde wird der Wert in einer Arrayliste gespeichert und die globale Zähl-Variable auf 0 zurückgesetzt. Nach einer Minute wird, mit den Werten aus der Arrayliste, die durschnittliche Anzahl der BpS in einer Minute berechnet. Anschließend wird die durchschnittliche Anzahl der BpS pro Minute in einer Textdatei gespeichert. In Abbildung 6.2 werden die gespeicherten Daten dargestellt. In Abbildung 6.3 sind die durchschnittlichen BpS über die gesamte Dauer der Messung abgebildet.

6.2.1 Messung

Insgesamt sind fünf Messungen, mit jeweils fünf Minuten, durchgeführt worden. Bei jeder Messung war eine unterschiedliche Anzahl von POIs geladen. Die Messungen sind alle mit einem Motorola Moto X der ersten Generation durchgeführt worden. Während jeder Messung waren in AREA regelmäßig POIs im Sichtfeld. Abbildung 6.2 zeigt, dass je mehr POIs geladen sind, die durchschnittliche Anzahl BpS pro Minute sinkt. Bei 10 und 25 geladenen POIs sind die BpS nahezu konstant bei 60. Wird die Anzahl der POIs auf 50 erhöht, dann sinken die durchschnittlichen BpS pro Minute leicht. Bei 75 POIs schwanken die Werte der durschnittlichen Anzahl BpS pro Minute stark. Sind 100 POIs geladen, dann ist die durchschnittliche Anzahl BpS pro Minute nahzu konstant bei 40. Die durchschnittliche Anzahl BpS (über die gesamte Dauer einer Messung) sinkt bei einer steigenden Anzahl geladener POIs (siehe Abbildung 6.3).

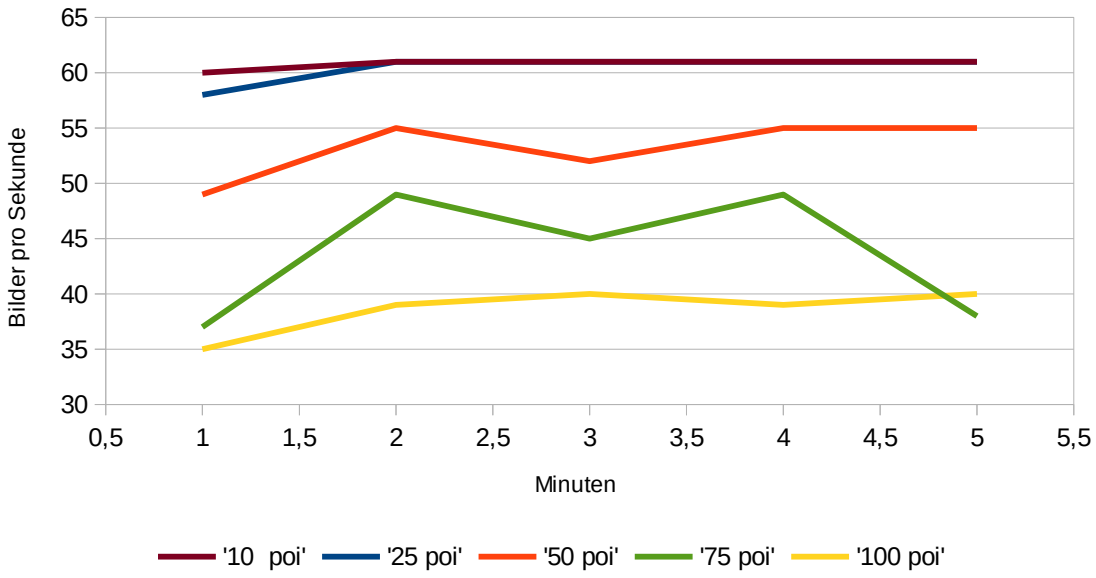

Abbildung 6.2: Messung der durchschnittlichen BpS über eine Minute, bei unterschiedlicher Anzahl POIs.

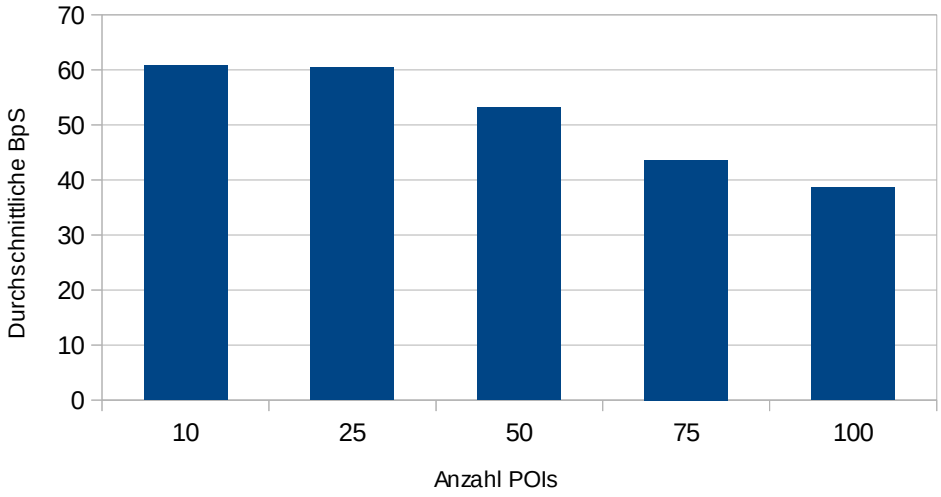

Abbildung 6.3: Durchschnittliche Anzahl BpS über eine Dauer von fünf Minuten, bei einer unterschiedlichen Anzahl geladener POIs.

6.2.2 Auswertung

In der ersten Minute sind die durchschnittlichen BpS pro Minute bei allen Messungen geringer als in den Minuten zwei bis fünf. Um ein POI zeichnen zu können, müssen mehrere Objekte angelegt werden. Diese müssen am Anfang in der Zeichenmethode initialisiert werden. Das Initialisieren eines POI-Objekts benötigt Zeit. Das ist der Grund dafür, dass die durchschnittlichen Anzahl BpS in der ersten Minute geringer ist, als in den Minuten danach.

Die Messungen werden von unterschiedlichen Einflüssen verfälscht. AREA kann nicht bei jeder Messung exakt gleich benutzt werden. Wird das Smartphone beispielsweiße in einer Messung öfters gedreht, dann müssen alle POIs ebenfalls gedreht werden. Bei einer Drehung sinkt die Anzahl der gezeichneten BpS und somit auch die durchschnittliche Anzahl BpS pro Minute. Die Messungen zeigen, dass die durchschnittliche Anzahl BpS pro Minute, mit zunehmender Anzahl POIs, sinkt.

In der AREA Version ohne OpenGL ES ist die Benutzeroberfläche aus mehreren Layouts zusammengesetzt. Diese Layouts enthalten eigene grafische Elemente. Die eigenen grafischen Elemente leiten von der Klasse *View* ab und überschreiben die Methode

onDraw(Canvas canvas). Alle *onDraw(Canvas canvas)* Methoden werden in einer unbekannten Reihenfolge aufgerufen. Deshalb ist es nicht möglich die BpS Anzahl zu ermitteln.

7

Fazit

Dieses Kapitel gibt zum Abschluss eine Zusammenfassung und einen Ausblick für mögliche AREA Erweiterungen. Dabei werden die größten Probleme und Herausforderungen noch einmal diskutiert.

7.1 Probleme und Herausforderungen

OpenGL ES ist eine Low-Level-Grafikbibliothek, welche nur grundlegende Funktionen zur Verfügung stellt. Mit OpenGL können nur grafische Primitive, wie beispielsweise Dreiecke, Punkte und Linien dargestellt werden. Das Ziel dieser Abschlussarbeit ist es, die Benutzeroberfläche von AREA mit grafischen Primitiven aus OpenGL ES zu konstruieren. Im Rahmen dieser Abschlussarbeit ist GraphyFrame entwickelt worden. GraphyFrame ist ein Grafik-Framework für die Darstellung von grafischen Elementen

mit OpenGL ES. Mit dem Einsatz von GraphyFrame kann mit einem Methodenaufruf beispielweise ein POI oder ein Radar dargestellt werden.

Die Basis von OpenGL ist die Grafikpipeline. Die Implementierung einer Grafikpipeline unterscheidet sich in OpenGL ES 1.1 und OpenGL ES 2.0. Die Grafikpipelines sind in OpenGL ES 1.1 und OpenGL ES 2.0 nicht untereinander kompatibel. Vor der Entwicklung von GraphyFrame mussten deshalb Vor- und Nachteile von beiden Implementierungen berücksichtigt werden. Mit dem Einsatz von OpenGL ES 1.1 in GraphyFrame können alle Anforderungen für die AREA Benutzeroberfläche erfüllt werden.

Eine Problem bei der Entwicklung von GraphyFrame ist das Multithreading. Die Liste *allRenderer* aus der Klasse *WorldRenderer* erhält Zugriffe von zwei unterschiedlichen Threads. Der Zeichenthread iteriert in einer Endlosschleife über die Liste *allRenderer*. Ein anderer Thread von AREA fügt der Liste *allRenderer* neue Elemente hinzu. Ein gleichzeitiger Zugriff aus zwei Threads kann eine *ConcurrentModificationException* auslösen. Wenn man den Zeichenthread regelmäßig anhalten würde, dann gäbe es Verzögerungen in der Benutzeroberfläche. Das Problem eine *ConcurrentModificationException* zu vermeiden, ist mit Hilfe von einem Bufferkonzept in der Klasse *WorldRenderer* gelöst worden.

Ein weiteres Problem ist die Messung von den BpS in der AREA Version ohne OpenGL ES. In der AREA Version ohne OpenGL ES ist die Benutzeroberfläche aus mehreren Layouts und grafischen Elementen zusammengesetzt. Die Zeichen-Methoden von den grafischen Komponenten werden in einer unbekannten Reihenfolge aufgerufen. Es ist nicht möglich die BpS zu messen.

Der Einsatz von GraphyFrame in AREA ermöglicht es die Benutzeroberfläche ohne Verzögerungen darzustellen. Zusätzlich verbraucht AREA mit GraphyFrame weniger Strom. Die Benutzeroberfläche aus geometrischen Primitiven mit OpenGL ES zu konstruieren, erfordert im Gegensatz zu einer Implementierung mit Android mehr Aufwand. Mit OpenGL ES kann die AREA Benutzeroberfläche in den folgenden Versionen mit dreidimensionalen grafischen Elementen erweitert werden.

7.2 Ausblick

In dieser Abschlussarbeit wird mit Hilfe von OpenGL ES eine zweidimensionale Benutzeroberfläche gezeichnet. Mit OpenGL ist es ebenfalls möglich dreidimensional zu zeichnen. Der Einsatz von einer dreidimensionalen Benutzeroberfläche in AREA ist vorstellbar. Beispielsweise kann der Kreis bei einem POI durch einen dreidimensionalen Pfeil oder durch eine Kugel ersetzt werden. Ein POI besteht bisher aus einem roten Kreis und einer Textbox mit der passenden Information. Der rote Kreis könnte durch ein Bild ersetzt werden und die Kategorie eines POIs anzeigen. Die Farben sind in GraphyFrame für die grafischen Primitiven frei wählbar. Mit den frei wählbaren Farben können unterschiedliche Farbschemata für AREA erstellt werden.

POIs, welche nahe zusammenliegen, überlagern sich in der AREA Version mit OpenGL ES immer noch. In der Abschlussarbeit von Julia Müller (vergleiche [Mül14]) werden Lösungsansätze zur Clusterbehandlung vorgeschlagen. Die Lösungsansätze aus [Mül14] können auch in die AREA Version mit OpenGL ES implementiert werden.

Abbildungsverzeichnis

1.1 Google Sky Map ist eine mobile AR-Anwendung. Es wird der Nachthimmel analysiert und Informationen eingeblendet [Dev]. 2

2.1 Definition eines roten Dreiecks. 6
2.2 Datenübertragung der Vertices von der Dalvik VM in die native Umgebung. In Anlehnung an [Kev13]. 7
2.3 Abstrakter Aufbau der OpenGL ES 1.1 Rendering Pipeline [ZG12]. 8
2.4 Orthogonale Projektion vs. perspektivische Projektion [WLH07] 10
2.5 Sichtvolumen bei orthogonaler Projektion und perspektivischer Projektion. In Anlehnung an [AMHH08]. 11
2.6 Texturkoordinaten sind unabhängig von der Textur und immer auf dem Intervall $[0, 1]$ definiert [SV12]. 12
2.7 Ein Texturfilter hat Auswirkungen auf die Qualität. 13

3.1 Grafische Oberfläche von AREA ohne OpenGL ES. 16

4.1 Die wichtigsten Komponenten der Architektur von GraphyFrame. 20
4.2 Das Prinzip der Klasse *Camera2D*. 23
4.3 Initialisierung der Projektionsmatrix. 24
4.4 Initialisierung der Projektionsmatrix für das Prinzip der Kamera. 25
4.5 Unterschiedliche Zeiger im Speicherbereich eines Vertice-Arrays. Der Speicherbereich enthält ein farbiges Dreieck. In Anlehnung an [ZG12]. . . 26

4.6 Unterschiedliche Zeiger im Speicherbereich eines Vertice-Arrays. Der Speicherbereich enthält ein Rechteck mit einer Textur. In Anlehnung an [ZG12]. 27
4.7 Ein Rechteck bestehend aus zwei Dreiecken. 28
4.8 Theoretischer Aufbau eines Kreises mit Vertices. 29
4.9 Beschreibung einer Matrix mit Matrixoperationen und die Integration in den Model-View-Matrix Stapel. 31
4.10 Aufrufhierarchie der Methoden pro gezeichnetem Frame 32
4.11 Aufbau eines beispielhaften Textur Atlas mit verschiedenen Zeichen. . . . 33
4.12 Prinzip des *locationView*-Objekts [GSP+14]. 37
4.13 Vererbungshierarchie des Radars ohne GraphyFrame (links) und mit GraphyFrame (rechts). 38
4.14 Architektur von AREA mit GraphyFrame. 39

5.1 Grafische Oberfläche von AREA mit OpenGL ES. 42

6.1 Messung des Stromverbrauchs bei einer variablen Anzahl von POIs. . . . 47
6.2 Messung der durchschnittlichen BpS über eine Minute, bei unterschiedlicher Anzahl POIs. 48
6.3 Durchschnittliche Anzahl BpS über eine Dauer von fünf Minuten, bei einer unterschiedlichen Anzahl geladener POIs. 49

Tabellenverzeichnis

5.1 Abgleich der funktionalen Anforderungen. (1) 42
5.2 Abgleich der funktionalen Anforderungen. (2) 43
5.3 Abgleich der nicht-funktionalen Anforderungen. (1) 44

Literaturverzeichnis

[AMHH08] AKENINE-MÖLLER, Tomas ; HAINES, Eric ; HOFFMAN, Natty: *Real-Time Rendering 3rd Edition*. Natick, MA, USA : A. K. Peters, Ltd., 2008. – 1045 S. – ISBN 987–1–56881–424–7

[API] API, Android: *Canvas*. `http://developer.android.com/reference/android/graphics/Canvas.html`, . – Zuletzt besucht: 10. Januar 2015

[Azu97] AZUMA, Ronald T.: A survey of augmented reality. In: *Presence* 6 (1997), Nr. 4, S. 355–385

[BR05] BIMBER, Oliver ; RASKAR, Ramesh: *Spatial Augmented Reality: Merging Real and Virtual Worlds*. Natick, MA, USA : A. K. Peters, Ltd., 2005. – ISBN 1568812302

[Dev] DEVS, Sky M.: *Google Sky Map Application*. `https://play.google.com/store/apps/details?id=com.google.android.stardroid&hl=de`, . – Zuletzt besucht: 10. Januar 2015

[ESV] *Choosing an OpenGL API Version*. `http://developer.android.com/guide/topics/graphics/opengl.html`, . – Zuletzt besucht: 26. Oktober 2014

[GPSM14] GINSBURG, D. ; PURNOMO, B. ; SHREINER, D. ; MUNSHI, A.: *OpenGL ES 3. 0 Programming Guide*. Addison Wesley Professional, 2014. – ISBN 9780321933881

[Gro] GROUP, Khronos: *OpenGL ES 1.1 Reference Pages*. https://www.khronos.org/opengles/sdk/1.1/docs/man/, . – Zuletzt besucht: 30. Dezember 2014

[GSP+14] GEIGER, Philip ; SCHICKLER, Marc ; PRYSS, Rüdiger ; SCHOBEL, Johannes ; REICHERT, Manfred: Location-based Mobile Augmented Reality Applications: Challenges, Examples, Lessons Learned. In: *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, 2014, 383–394

[HP96] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture: A Quantitative Approach 2nd Edition.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1996. – ISBN 1–55860–329–8

[Kev13] KEVIN, Brothaler: *OpenGL ES 2 for Android: a quick-start guide*. Pragmatic Bookshelf, 2013. – ISBN 9781937785345

[KM13] KOMATINENI, Satya ; MACLEAN, Dave: *Expert Android*. 1st. Berkely, CA, USA : Apress, 2013. – ISBN 1430249501, 9781430249504

[lea] *Learn OpenGL ES*. http://www.learnopengles.com/android-lesson-six-an-introduction-to-texture-filtering/, . – Zuletzt besucht: 24. Oktober 2014

[Mil] MILLER, John: *How to Convert from RGB 255 to OpenGL Float Color*. https://www.opengl.org/discussion_boards/showthread.php/139274-How-to-Convert-from-RGB-255-to-OpenGL-Float-Color, . – Zuletzt besucht: 28. Oktober 2014

[Mül14] MÜLLER, Julia: *Konzeption und prototypische Implementierung eines Verfahrens zur POI Clusterbehandlung innerhalb einer Augmented Reality Anwendung*. http://dbis.eprints.uni-ulm.de/1062/. Version: 2014. – Zuletzt besucht: 17. November 2014

[NFHS12] NISCHWITZ, A. ; FISCHER, M. ; HABERÄCKER, P. ; SOCHER, G.: *Computergrafik und Bildverarbeitung: Band I: Computergrafik*. Vieweg+Teubner

Verlag, 2012. – ISBN 9783834883230

[Soo12] SOOD, Raghav: *Pro Android Augmented Reality*. 1st. Berkely, CA, USA : Apress, 2012. – ISBN 143023945X, 9781430239451

[SSKLK13] SHREINER, Dave ; SELLERS, Graham ; KESSENICH, John M. ; LICEA-KANE, Bill M.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. 8th. Addison-Wesley Professional, 2013. – ISBN 0321773039, 9780321773036

[SSP+13] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; NIENHAUS, Hans ; REICHERT, Manfred: Using Vital Sensors in Mobile Healthcare Business Applications: Challenges, Examples, Lessons Learned. In: *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, 2013, 509–518

[SSP+14] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; MAIER, Fabian ; REICHERT, Manfred: Towards Process-Driven Mobile Data Collection Applications: Requirements, Challenges, Lessons Learned. In: *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, 2014, 371–382

[SV12] SMITHWICK, Mike ; VERMA, Mayank: *Pro OpenGL ES for Android*. 1st. Berkely, CA, USA : Apress, 2012. – ISBN 1430240024, 9781430240020

[TCG+08] TAKACS, Gabriel ; CHANDRASEKHAR, Vijay ; GELFAND, Natasha ; XIONG, Yingen ; CHEN, Wei-Chao ; BISMPIGIANNIS, Thanos ; GRZESZCZUK, Radek ; PULLI, Kari ; GIROD, Bernd: Outdoors augmented reality on mobile phone using loxel-based visual feature organization. In: *Proceedings of the 1st ACM international conference on Multimedia information retrieval* ACM, 2008, S. 427–434

[Wik13] WIKIBOOKS: *OpenGL Programming/Modern OpenGL Tutorial Text Rendering 02 — Wikibooks, The Free Textbook Project*. `http://en.wikibooks.org/w/index.php?title=OpenGL_Programming/`

Modern_OpenGL_Tutorial_Text_Rendering_02&oldid=2580590. Version: 2013. – Zuletzt besucht: 5. Januar 2015

[WLH07] WRIGHT, Richard ; LIPCHAK, Benjamin ; HAEMEL, Nicholas: *Opengl® Superbible: Comprehensive Tutorial and Reference, Fourth Edition*. Fourth. Addison-Wesley Professional, 2007. – ISBN 9780321498823

[ZDB08] ZHOU, Feng ; DUH, Henry Been-Lirn ; BILLINGHURST, Mark: Trends in augmented reality tracking, interaction and display: A review of ten years of ISMAR. In: *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality* IEEE Computer Society, 2008, S. 193–202

[ZG12] ZECHNER, Mario ; GREEN, Robert: *Beginning Android Games*. 2nd. Berkely, CA, USA : Apress, 2012. – ISBN 1430246774, 9781430246770

Name: Uwe Ullrich Matrikelnummer: 751328

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Uwe Ullrich